

**Missing Ones.**

**TreeMap Implementation**

**Union Find + Priority Queue**

Time  
Complexity

Space  
Complexity

Arrays &  
Recursion

Sorting &  
Searching

Dynamic  
Programming

List, Stack,  
Queues

Divide &  
Conquer

Graphs

Hashing

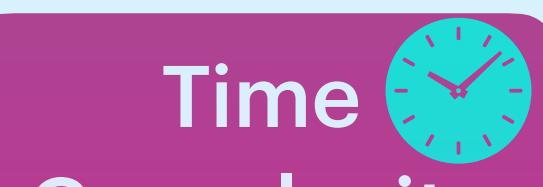
Trees,  
Trie (Advanced Trees)

Heaps

Greedy  
Algorithms

Solving System Design  
problems with DS & Algo





Time  
Complexity

Completed As of Today



Space  
Complexity



Sorting &  
Searching



Dynamic  
Programming



Arrays &  
Recursion



Divide &  
Conquer



Starting From Monday  
18th Oct 2021

List

Hashing

Stack, Queues

Graphs

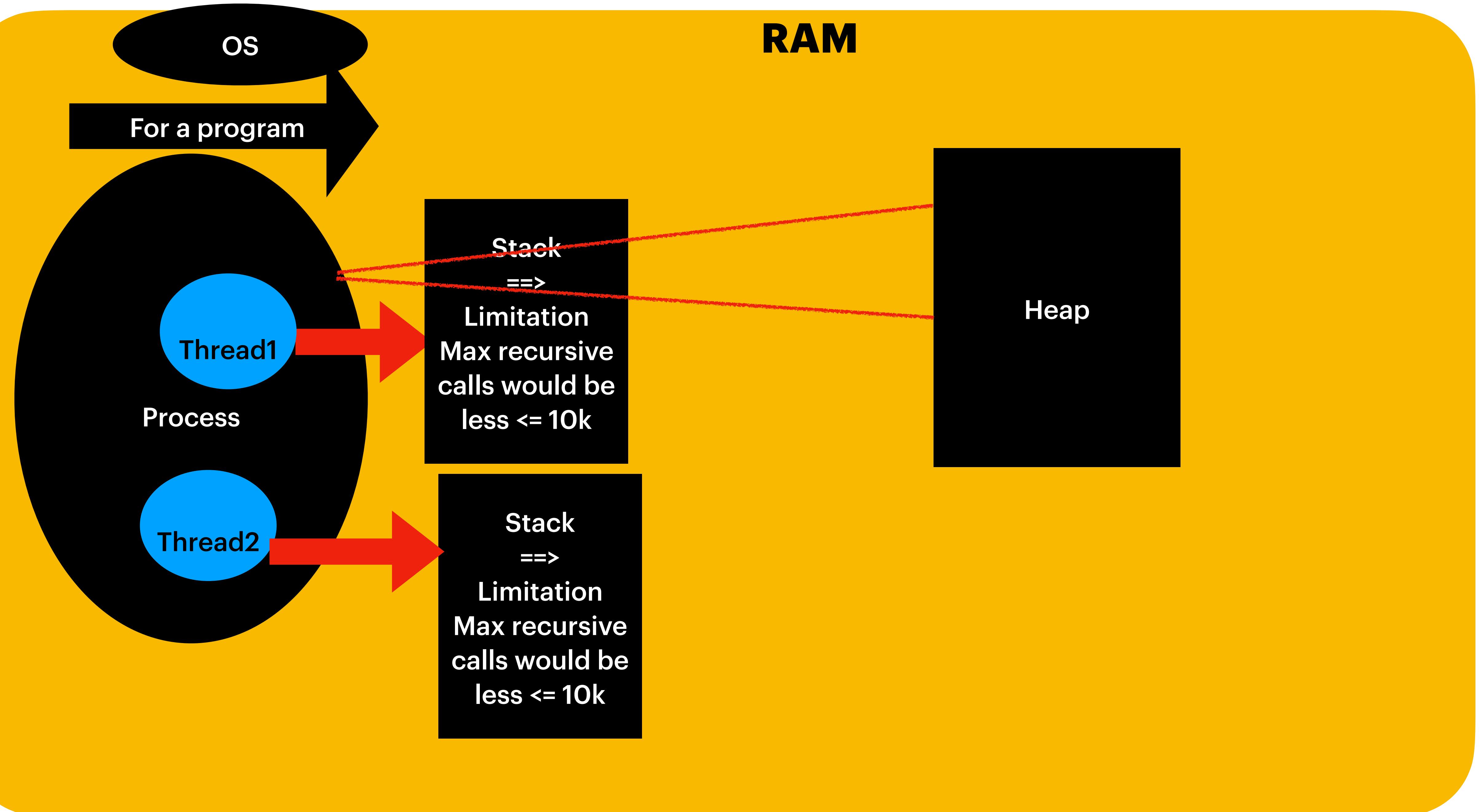
Math  
Patterns

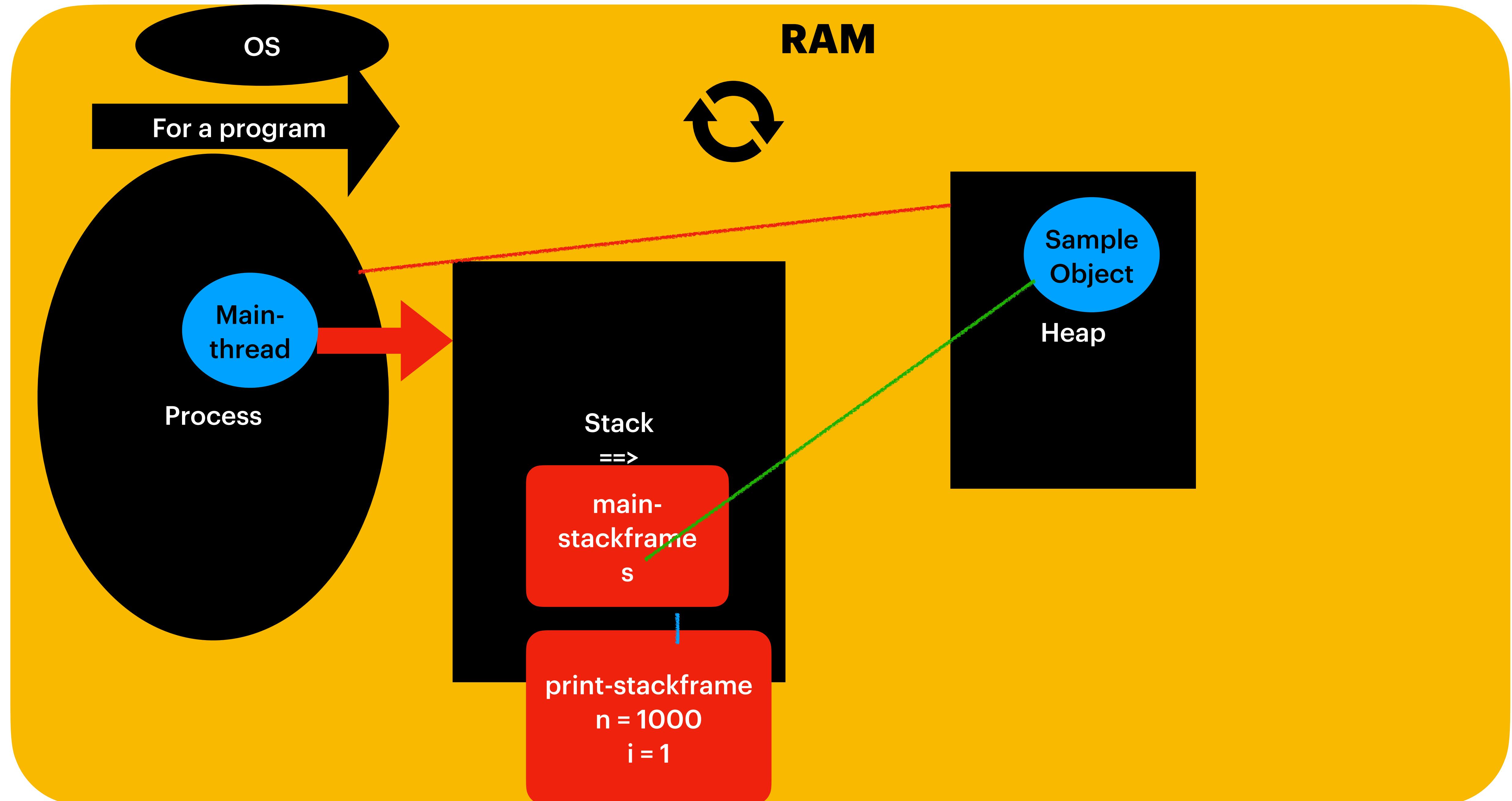
Solving System  
Design problems with DS

Greedy  
Algorithms

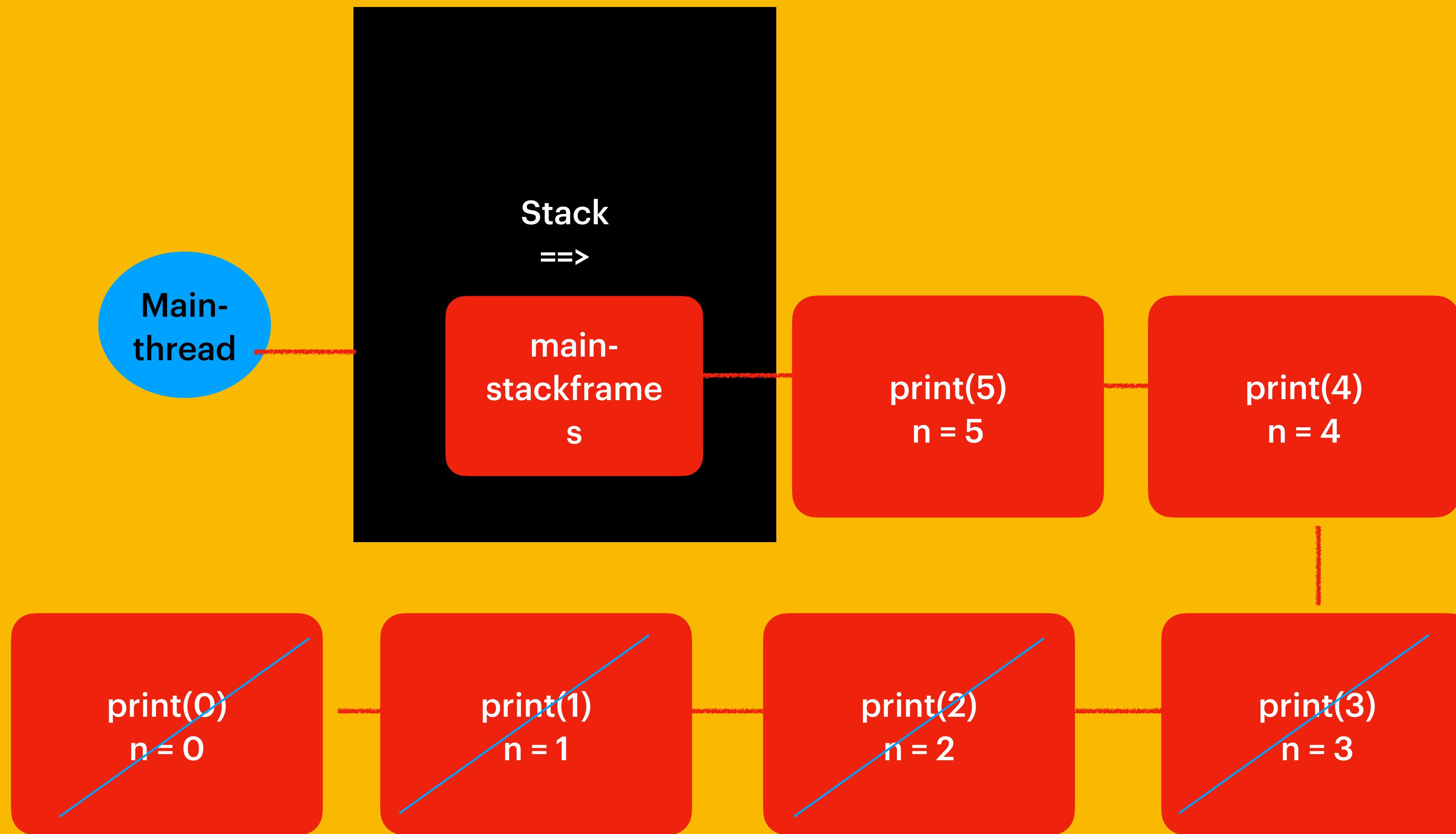
Trees,  
Trie (Advanced Trees)

Heaps



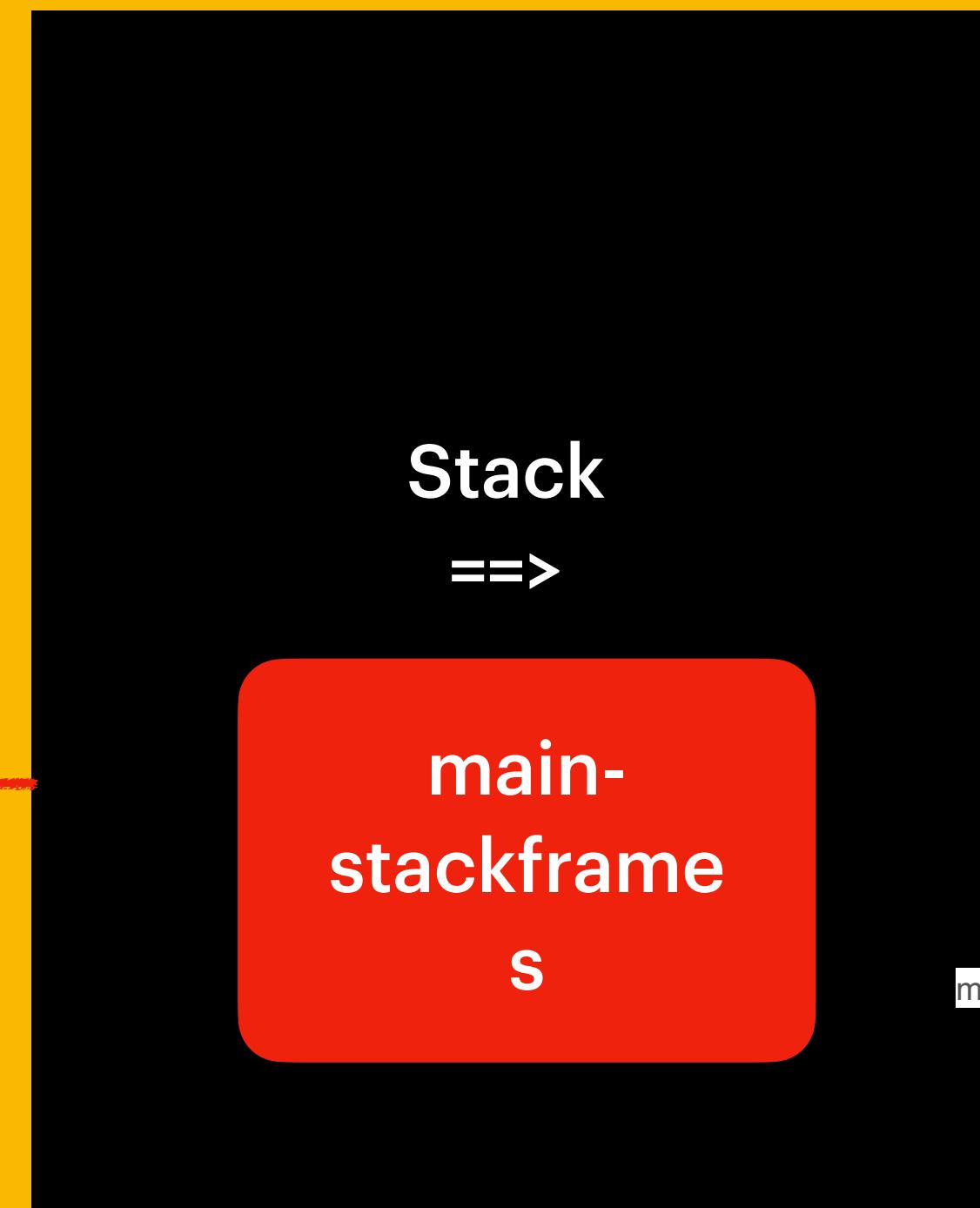


# RAM



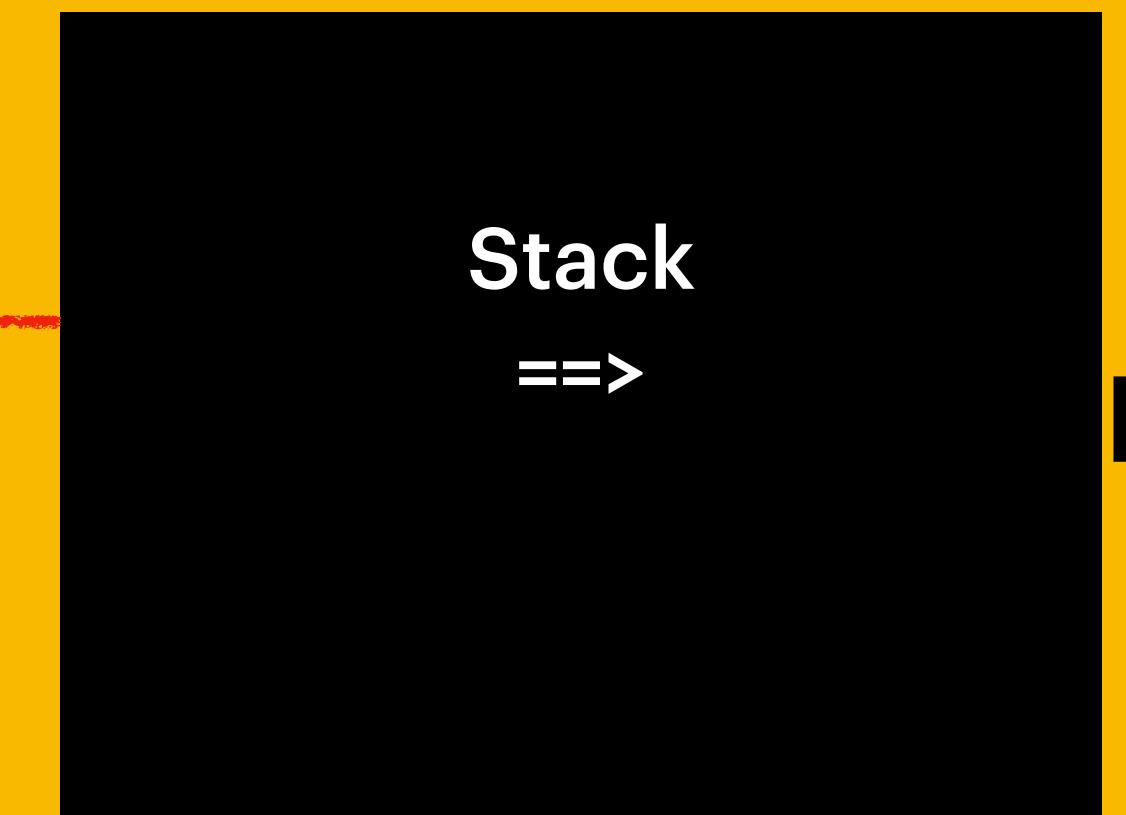
**RAM**

Main-thread



Print 10k numbers

User thread



Print another 10k numbers

**Memory Allocation  
(Space Complexity)**

**RAM**

**Data Processing  
(Time Complexity)**

## **Use case1 : Transfer/Share File**

**File size : 100MB**

**Time Complexity : 1 sec ~ = O(1sec)**

**Space Complexity : 100MB = O(100MB)**

**File size : 1GB**

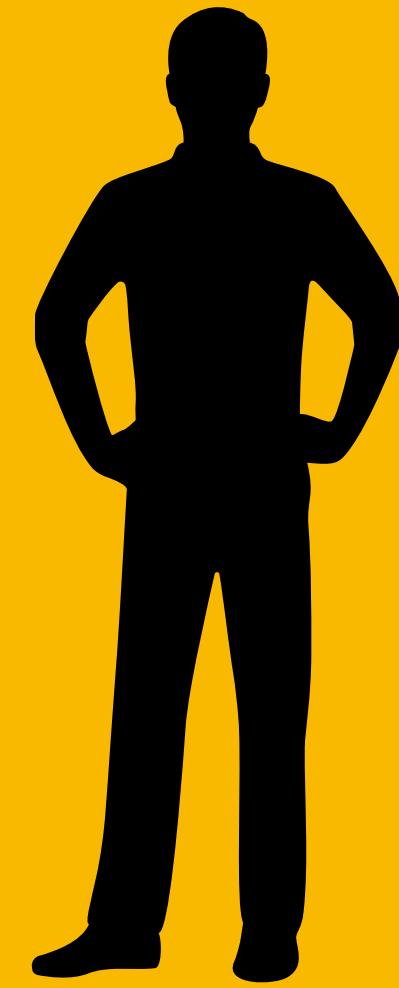
**Time Complexity : 10 sec ~ = O(10sec)**

**Space Complexity : 1024MB = O(1GB)**

**File size : 1PB**

**Time Complexity : = O(days)**

**Space Complexity : = O(1PB)**

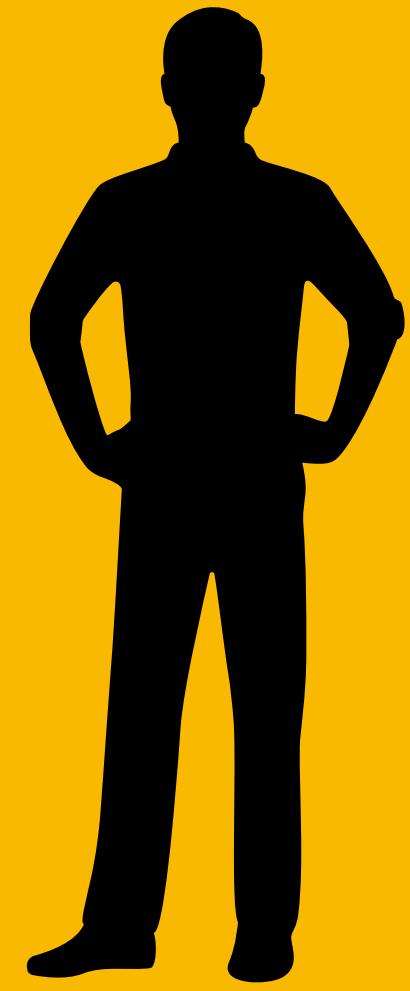


**Srinu Hyd (IND)**



**Kiran USA**

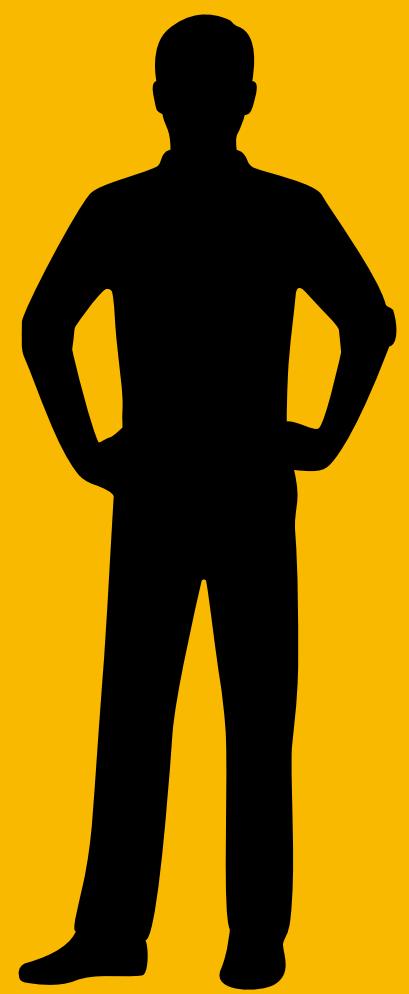
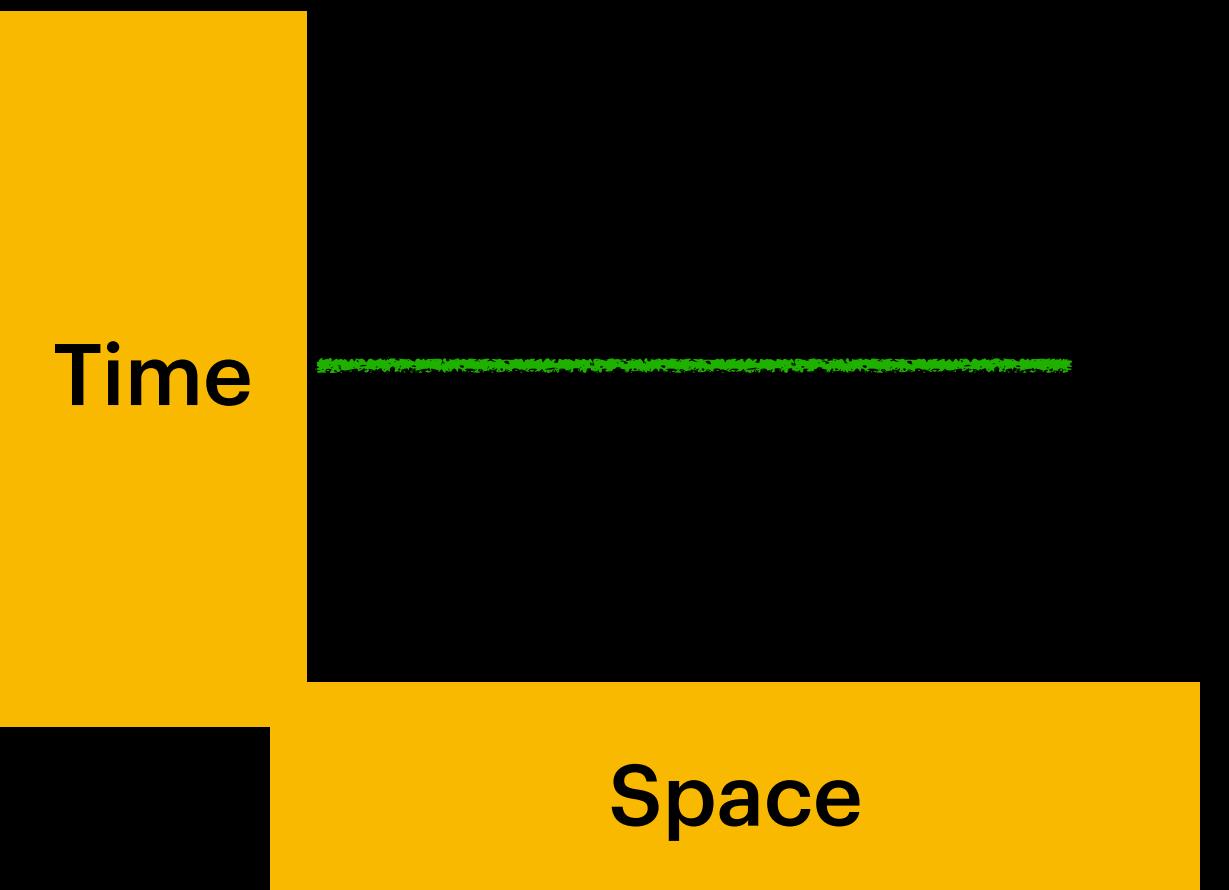
## Use case2 : Physical delivery



**Srinu Hyd (IND)**

On Flight =>

Time =>  $O(7\text{hours}) \sim O(1)$   
Space =>  $O(\text{FileSize}) \sim O(n)$

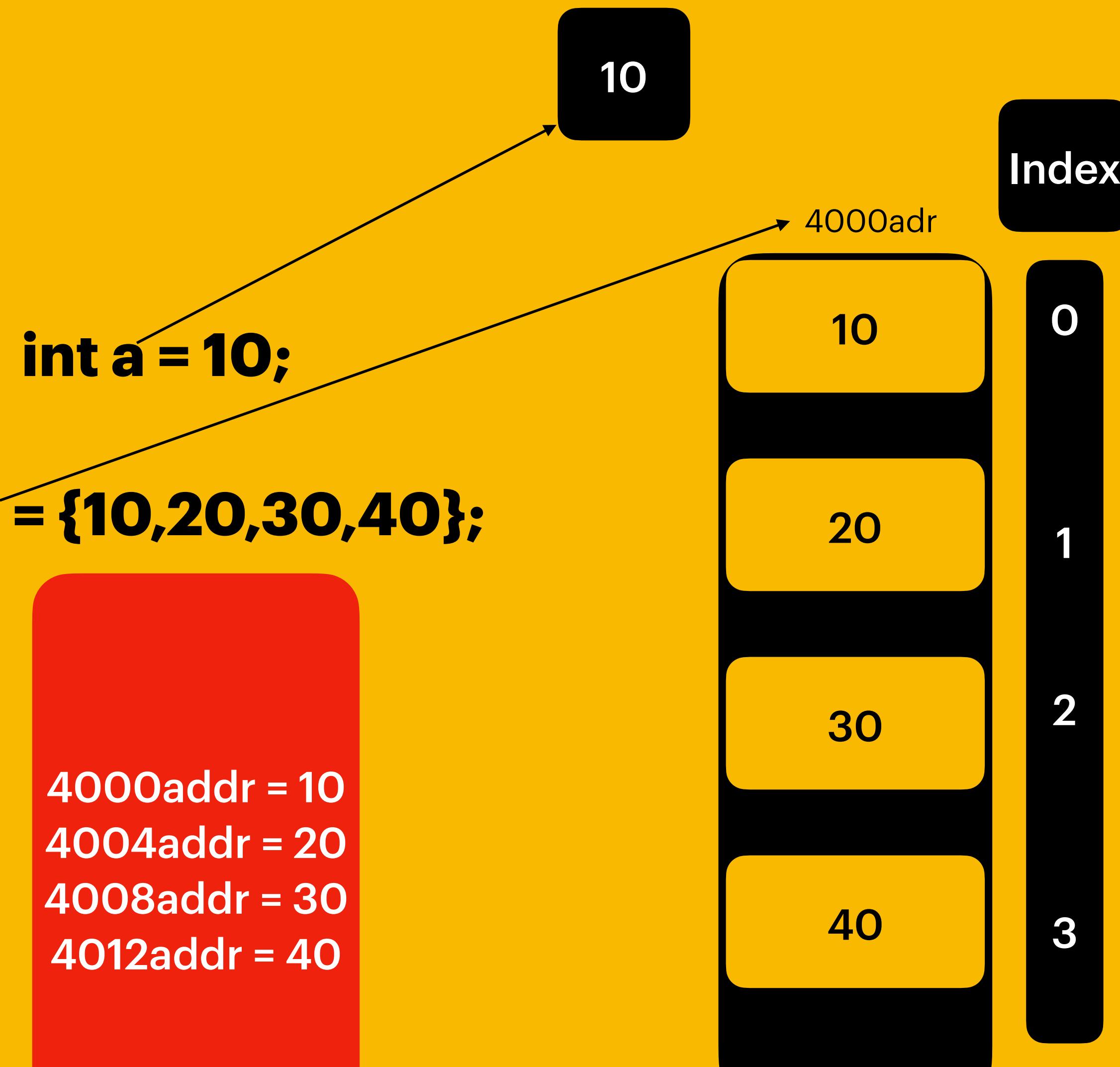


**Kiran USA**

```
a[0] = 4000 + 0*4 =  
4000addr = 10  
a[1] = 4000 + 1*4 =  
4004 add = 20
```

```
int[] a = {10,20,30,40};
```

```
4000addr = 10  
4004addr = 20  
4008addr = 30  
4012addr = 40
```



**For Data Access Array is recommend ::**

**Time Complexity :**

If we know the index number then its = O(1)

Otherwise Iteration is needed so O(n)

**Space Complexity : O(n)**

**Data Deletion =>**

**Time Complexity :**

when you delete an element  
all the elements should be  
moved an index before.

It's a costlier operation.

**Time Complexity : O(n)**

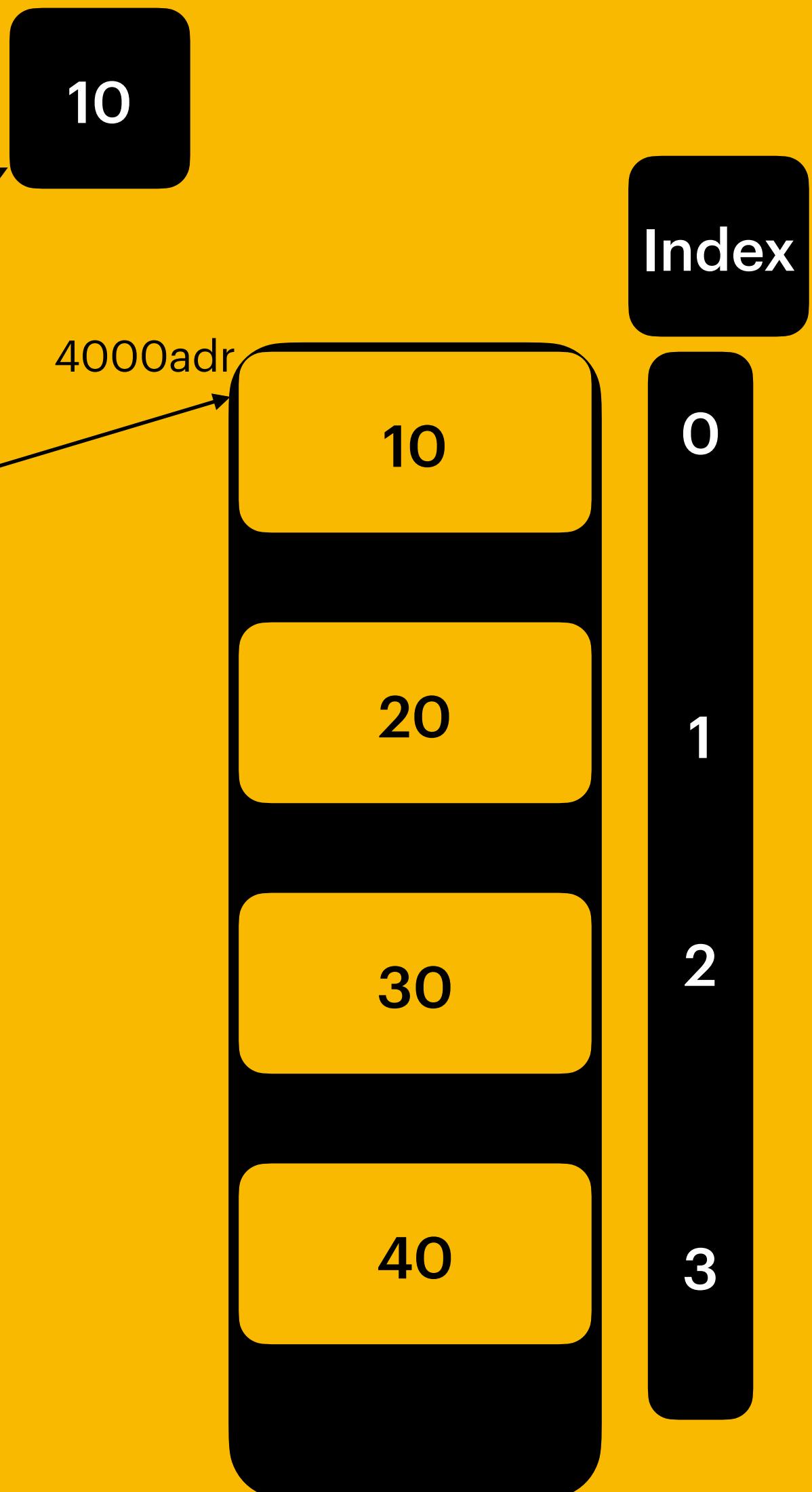
**For Data Deletion Array is  
not recommended.**

$$\begin{aligned}a[0] &= 4000 + \\0 * 4 &= 4000 = \\10 &\\a[1] &= 4000 + \\1 * 4 = 4004 \text{ add} &= 20\end{aligned}$$

$$\begin{aligned}4000\text{addr} &= 10 \\4004\text{addr} &= 20 \\4008\text{addr} &= 30 \\4012\text{addr} &= 40\end{aligned}$$

**int a= 10;**

**int[] a = {10,20,30,40};**



## Finally about Array :-

\*\*\*\*

- => Array is a data structure, it allows similar type of data.
- => Array stores more than one element.
- => In an Array elements stored based on index number.
- => Index number starts from 0 ends with n-1. Here n is the length of an array.

Ex:-

```
int[] count = { 10,20,30,40};  
count[0] returns 10  
count[1] returns 20  
etc..
```

### Accessing :

Time Complexity based on Index = O(1)  
Time Complexity to Search an Element = O(n)  
Space Complexity = O(n)

### Deletion :

Time Complexity :  
when you delete an element all the elements should be moved an index before.  
It's a costlier operation.  
Time Complexity : O(n)

For Data Deletion Array is not recommended.

## **Stack**

**Stack Frame :**

**Every method execution happens within a stack frame.**

**All the local variable will be initialised.**

**Your logic executes.**

**Stack returns value to the caller if return type presents .**

**Then finally Stack Frame will terminated.**

## Stack

main-stackframe

fact-stackframe  
n=3 ; result =1;  
Update values  
n= 1 result = 6

```
public int fact(int n)
{
    if(n <= 1)
    {
        return 1;
    }

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

main -> fact(3)

fact(3) => n = 3

return 3\* fact(2)

return 3\*2 = 6

fact(2) => n = 2

return 2\* fact(1)

return 2\*1 = 2;

fact(1) => n = 1  
if(n <= 1)  
return 1;

return 1;

## What is Recursion ?

Recursion is an approach to solving problems using a function that calls itself as a subroutine.

For Recursion base case is mandatory otherwise its leads to stack overflow.

Recursion is good approach when you derive optimised solution from sub problems.

## Limitations on Recursion :

1. Recursion takes O(n) Space as the separate stack frame is getting created for every recursive call.
2. Complex to understand , it would be difficult to update the recursion logic.

Recursion is good when you feel iterative approach for a given problem increases the complexity and compromise on space.

## **Why Sorting ?**

**Usually for searching an element in unsorted array takes linear time i.e  $O(n)$ .**

**If we sort the array, we can apply the binary search on elements so that searching would be done within  $O(\log n)$ .**

**That's a great improvement when we consider larger datasets.**

Sorted array has binary search tree property. If you take a mid element elements less than "mid" would be on left side & elements greater than "mid" would be on right side. This key property we can use to divide array to half in every iteration/recursive call

{1,2,3,4,**5**,6,7,8,9}. => find key = 1

As the mid element **5** > key i.e (1)

Take left part of array {1,**2**,3,4}. Exclude right side part {6,7,8,9}

As the mid element 2 > key i.e (1)

Take left part of array {**1**}. Exclude right side part {3,4}

As Mid element 1 == key (i.e 1)

Element found.

If size of the array is 8 => you can find the element in 3 steps

If we use recursion , the we could call method recursively logn times. So that there are logn stack frames would be active.

So In case of recursion the SpaceComplexity would be O(logn)

$n = 8$   
↓  
 $n/2$   
↓  
 $n/4$   
↓  
 $n/8$

In a binary search for every iteration/ recursion we divide array half. For n = 8 , n/8 is the possible sub problem so this can be reached in 3 steps .

$$\log_2^3 = 8$$

So the TimeComplexity is O(log(n))

{1,2,3,4,**5**,6,7,8,9}. => find key = 10

As the mid element **5** < key i.e (10)

Exclude left part of array {1,2,3,4}. Include right side part {6,**7,8,9**}

As the mid element **7** < key i.e (10)

Exclude left part of array {6}. Include right side part {**8,9**}

As Mid element **8** < key i.e (10)

Exclude left part of array {}. Include right side part {9}

Mid element 9 < 10

There are no elements in array to look so Element is not found.

**2 power 1 = 2**

**2 power 2= 4**

**2 power 3= 8**

.....

**2 power 15= 32,768**

.....

**2 power 31 = 2,147,483,648**

**Time Complexity => log n with base 2  
=> O(log n)**

**When we working on Sorting the following techniques would be considered:**

1. Time Complexity
2. Space Complexity
3. Stability
4. Internal Sort / External Sort
5. Recursive / Non-Recursive
6. Compare / Swap

## Time Complexity :

The easiest way to classify an algorithm is by *time complexity*, or by how much relative time it takes to run the algorithm

## Stability :

Stability talks about , there can be duplicate elements in an array. After sorting would that order be maintained.

Ex : arr : { **1**,7,2,**1**} Here we have element 1 as duplicate with sort property as colour. So after sort Red colour 1 should come first then the Green 1 should appear.

After sort : arr : { **1**,**1**,7,2}

## Space Complexity :

There are two types of classifications for the space complexity of an algorithm:

*in-place* or *out-of-place* given a different input size.

*in-place* algorithm talks about swapping the elements in an original array, which results in constant space complexity. Its more space efficient but mis use can leads to data mess.

*Out-Place* algorithm always takes at the extra copy of data which leads  $O(n)$  space complexity.

## Internal (RAM) / External (Disk) Sort:

If sorting happens on RAM its internal sort , If the sort happened as hard disk level then its External Sort.

We usually go for External Sort if the input data size is greater than RAM size.

## Recursive / Non-Recursive :

Talks about Is the sort login is on iterative or recursive.

## Compare/Swap :

Talks about does the sort logic takes how may comparisons & swaps

### Selection Sort :

i = 0 {**5**, 11, 3, 2, 10, 1}

i = 1 {**1**, **11**, 3, 2, 10, 5}

i = 2 {**1**, **2**, **3**, 11, 10, 5}

i = 3 {**1**, **2**, **3**, **11**, 10, 5}

i = 4 {**1**, **2**, **3**, **5**, **10**, 11}

i = 5 {**1**, **2**, **3**, **5**, **10**, **11**}

Time Complexity =>  $n^2$

Space Complexity => O(1)

Stability => There is No Stability

Internal Sort

Non - Recursive

Swap => swap(n)

Comparison Sort => Yes

### Selection Sort Algorithm :

For current element iteration =>

Find out the smallest element if the element found then swap the current element with smallest element.

Repeat the iteration process from index 0 to n-1 .

In the example **pink** colour represents current element , **green** colour represents smallest element left in array so that both will be swapped.

**SelectionSort** can be used when the elements are unsorted.

**TimeComplexity =  $O(n^2)$**

**Space Complexity =  $O(1)$**

**Swap =  $O(n)$**

**Selection Sort gives best performance when you are using smaller set of array (size  $\leq 50$ )**

**This would never happens in real time.**

**Another draw back of Selection Sort is, it takes  $O(n^2)$  time complexity even the elements in array are sorted.**

## Insertion Sort :

i = 1 {1,**2**, 3,5}

Inner loop.

Do sorting only when  
currentIndex - 1 value is greater  
than current element.

Then repeats the process.

i = 2 {1,2, **3**,5}

i = 3 {1,2,3,**5**}

## Insertion Sort :

Insertion sort gives you best time complexity O(n) when the elements are sorted.

Insertion sort divides the given array into two subsets . Sorted and Unsorted.

Ex :

i = 1 {1,**2**, 3,5} Here current integration is 1 so insertion sort make sure that left part of the current iteration to be sorted.  
{1,**2** ....}

Insertion Sort do the sorting only when the left part element is greater than current interaction element so that when the array is already sorted inner loop always executes in constant time.

For a sorted array overall time complexity => O(n)

If the array is not sorted then the insertion sort would give the time complexity as => O( $n^2$ )

## Insertion Sort :

i = 1 {7,4,5,2}  
=> inner loop  
{7,4 ...}  
=> {4,7,...}

i = 2 { 4,7,5,2}  
=> inner loop  
{ 4,7,5 ....}  
J = i-1  
j = 1  
**while (j>=0 && arr[j] > 5)**  
  
{4,5,7,....}

i=3. { 4,5,7,2}

=> inner loop  
{ 4,5,7,2}  
J = i-1

**while (j>=0 && arr[j] > 2)**

j = 2 { 4,5,2,7}

j = 1 {4,2,5,7}

j=0 {2,4,5,7}

**TimeComplexity :**  
Sorted Array => O(n)  
Unsorted Array => O( $n^2$ )  
**Space Complexity => O(1)**  
**Stability => Stable**  
**Comparison Sort => Yes**  
**Swap => O( $n^2$ )**  
**NonRecursive**  
**Internal Sort**

If the input array has all unsorted elements then insertion sort is not recommended. It's not only takes time complexity  $O(n^2)$  even the swap happens  $n^2$  times.



Question & Answers on  
Sorting

## OutPlace Algorithm

`int[] arr = { 10,-5,12,-9,13,14,-1 } => {-5,-9,-1,.....}`

1 iteration to found how -ve elements present = 3 => O(n)

`int[] neg = new int[3];`

Iterate main array insert -ve elements -negarr. => O(n)

Time Complexity => O(n)

Space Complexity => O(n)

**InPlace Algorithm**  
**Time Complexity => O(n)**  
**Space Complexity => O(1)**

**int[] arr = { 10,-5,12,-9,13,14,-1 } => {-5,-9,-1,....}**

**=> startIndex = 0**

**Iteration from i = 1, move till n**

**{**

**=> arr[i] < 0**

**Swap => startIndex & current "i"**

**startIndex++;**

**}**

## Bubble Sort :

Bubble sort walks through the array, compares two elements at a time. If the elements are out of order, it swaps them. It continues to swap out of order elements until the entire collection is sorted.

Yes Bubble sort is an efficient algorithm, It takes Time Complexity as  $O(n^2)$  and does  $n^2$  swaps. It makes your algorithm run slow.

There are two notes on Bubble Sort :

=> For every iteration the largest element would be swapped .

Assume currently nth iteration is running by this time n-1 elements would already been swapped.

=> If the input array is sorted then there won't be any swapping in bubble sort, this key we can use to bypass next iterations.

By considering above two points still the Bubble sort can be used on demand.

## Bubble Sort :

i = 0 {**7,4,5,2**}  
Inner loop  
Sort j=0 ; j<arr.length;j++  
=> {4,7,5,2}  
=> {4,5,7,2}  
=> {4,5,2,7}

i = 1 {**4,5,2,7**}  
Inner loop  
Sort j=0 ; j<arr.length ;j++  
=> {4,5,2,7}  
=> {4,2,5,7}  
=> {4,2,5,7}

i = 2 {**4,2,5,7**}  
Inner loop  
Sort j=0 ; j<arr.length;j++  
=> {2,4,5,7}  
=> {2,4,5,7}  
=> {2,4,5,7}

i = 3 {**2,4,5,7**}  
Inner loop  
Sort j=0 ; j<arr.length;j++  
=> {2,4,5,7}  
=> {2,4,5,7}  
=> {2,4,5,7}

**TimeComplexity :**  
**Sorted Array => O(n)**  
**Unsorted Array => O( $n^2$ )**  
**Space Complexity => O(1)**  
**Stability => Stable**  
**Comparison Sort =>Yes**  
**Swap => O( $n^2$ )**  
**NonRecursive**  
**Internal Sort**

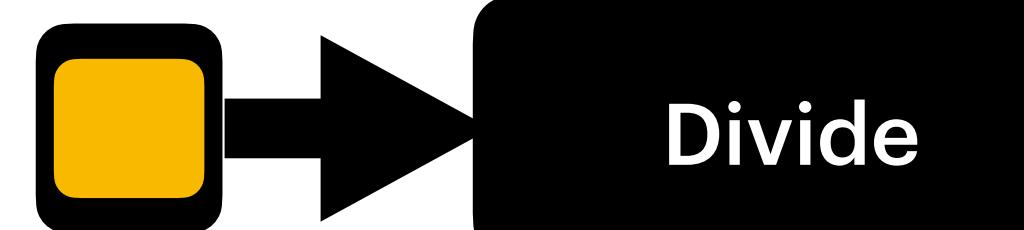
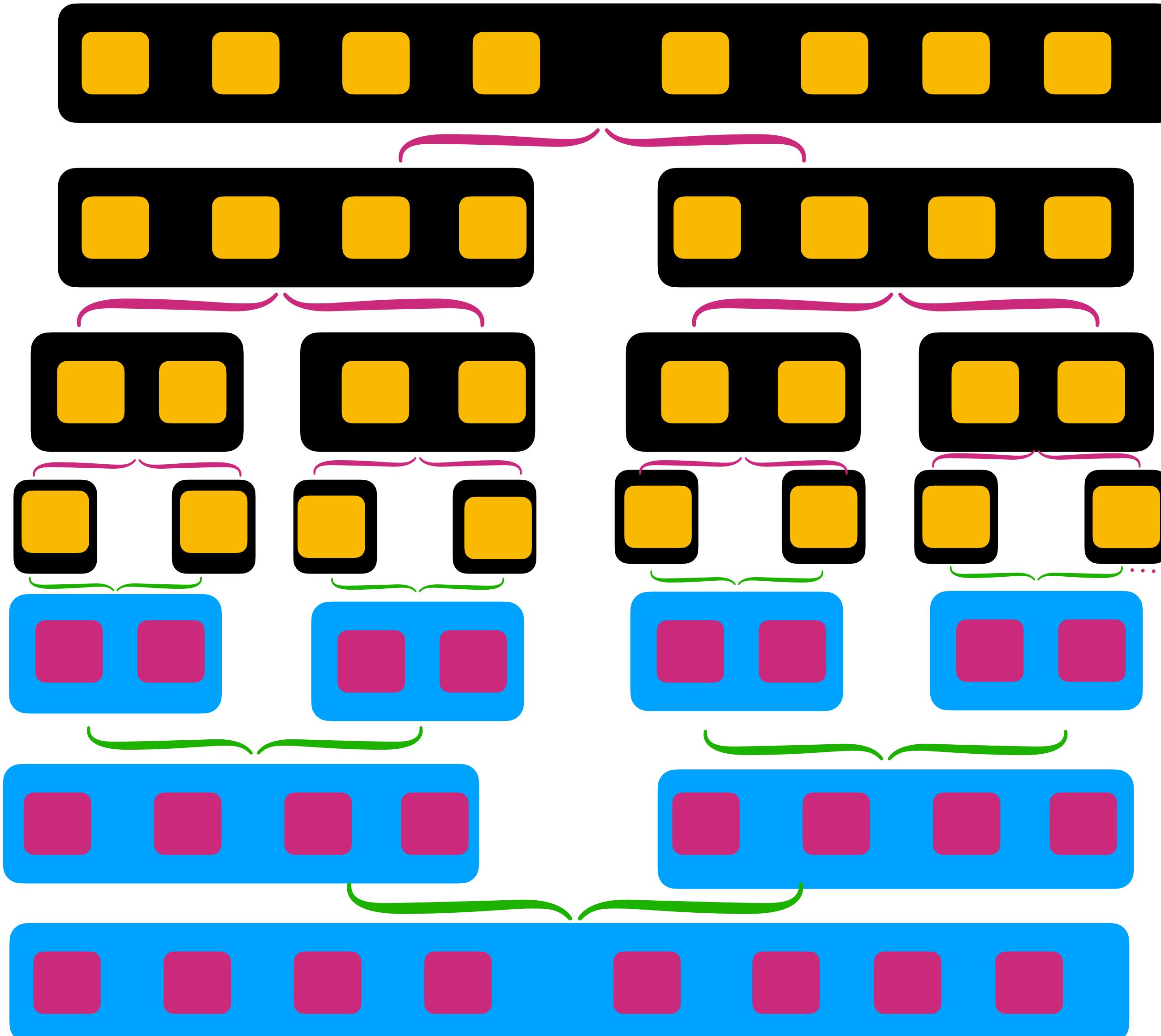
If the input array has all unsorted elements then Bubble sort is not recommended. It's not only takes time complexity  $O(n^2)$  even the swap happens  $n^2$  times.

Merge Sort uses divide and conquer pattern.

Merge sort divides the problem into possible small problems then applies sorting recursively.

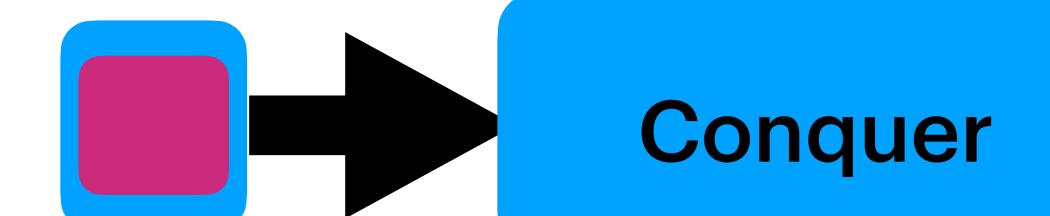
Divide => divides source collection into possible  $n/2$  sub problems recursively.

Conquer=> Applies the sorting at subproblem level (compare, swap & merge) then repeats recursively.

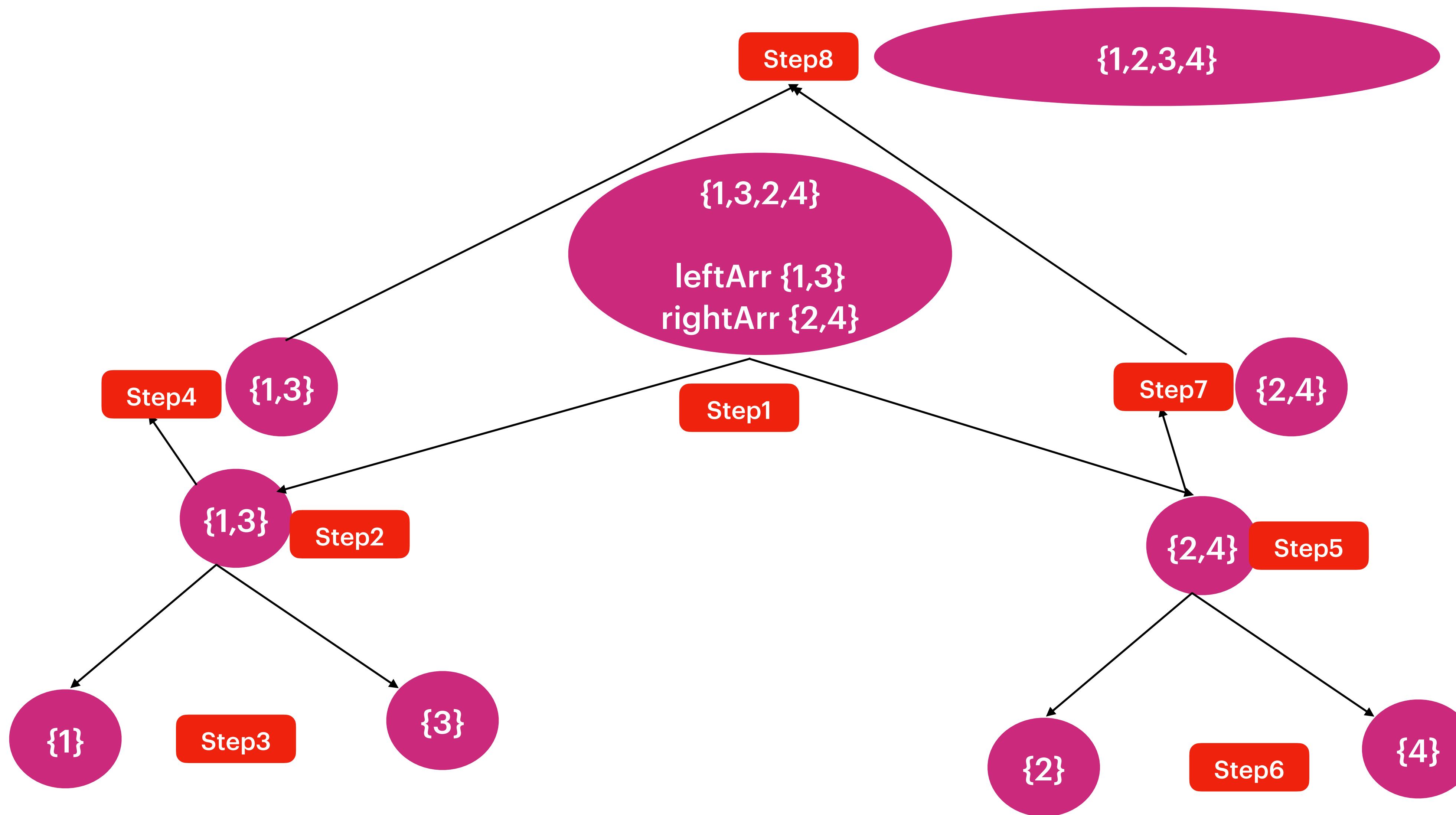


Divide=> Break up the problem into smallest possible sub problems.

Compare, Swap & Merge



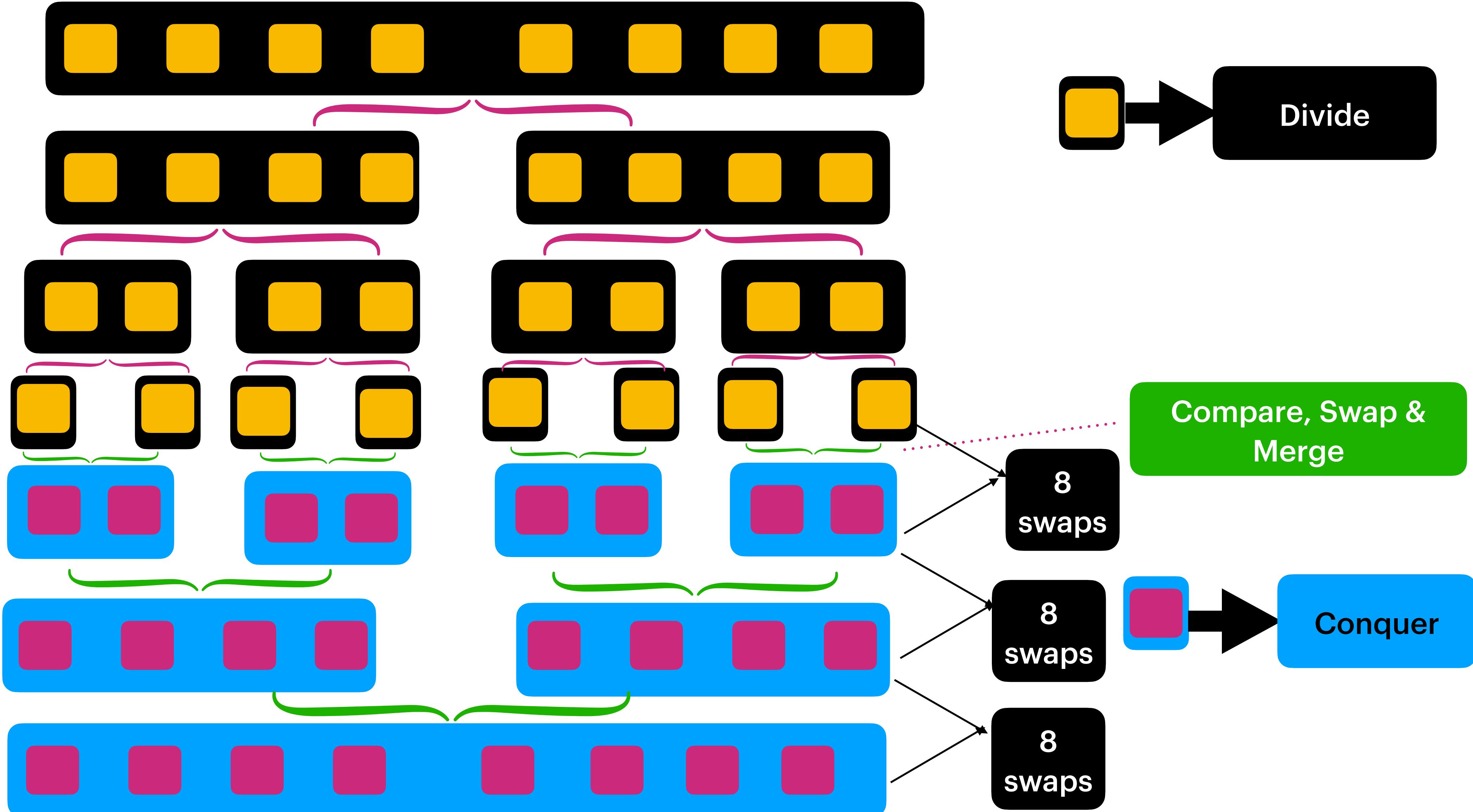
Conquer=> Figure out the solution for the smallest sub problem, then apply the same technique to solve larger problems recursively .



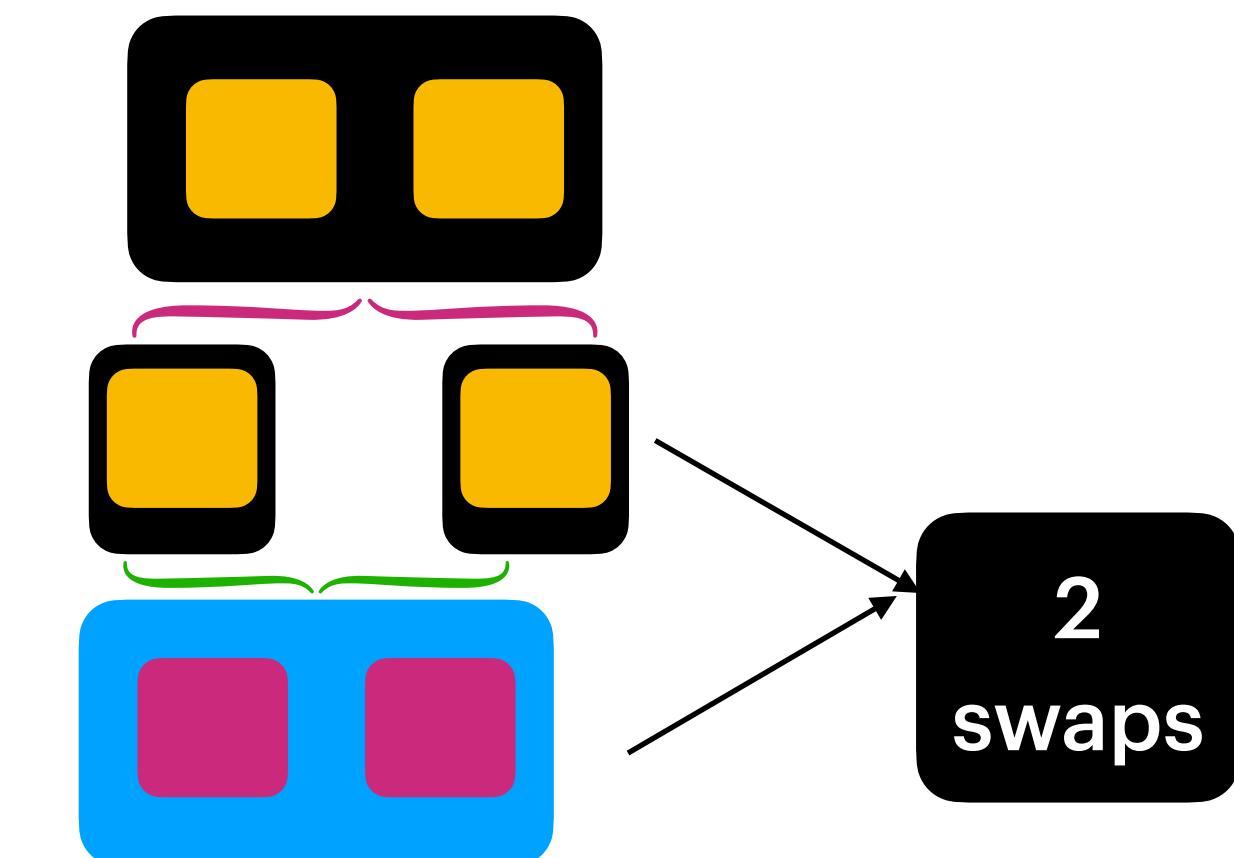
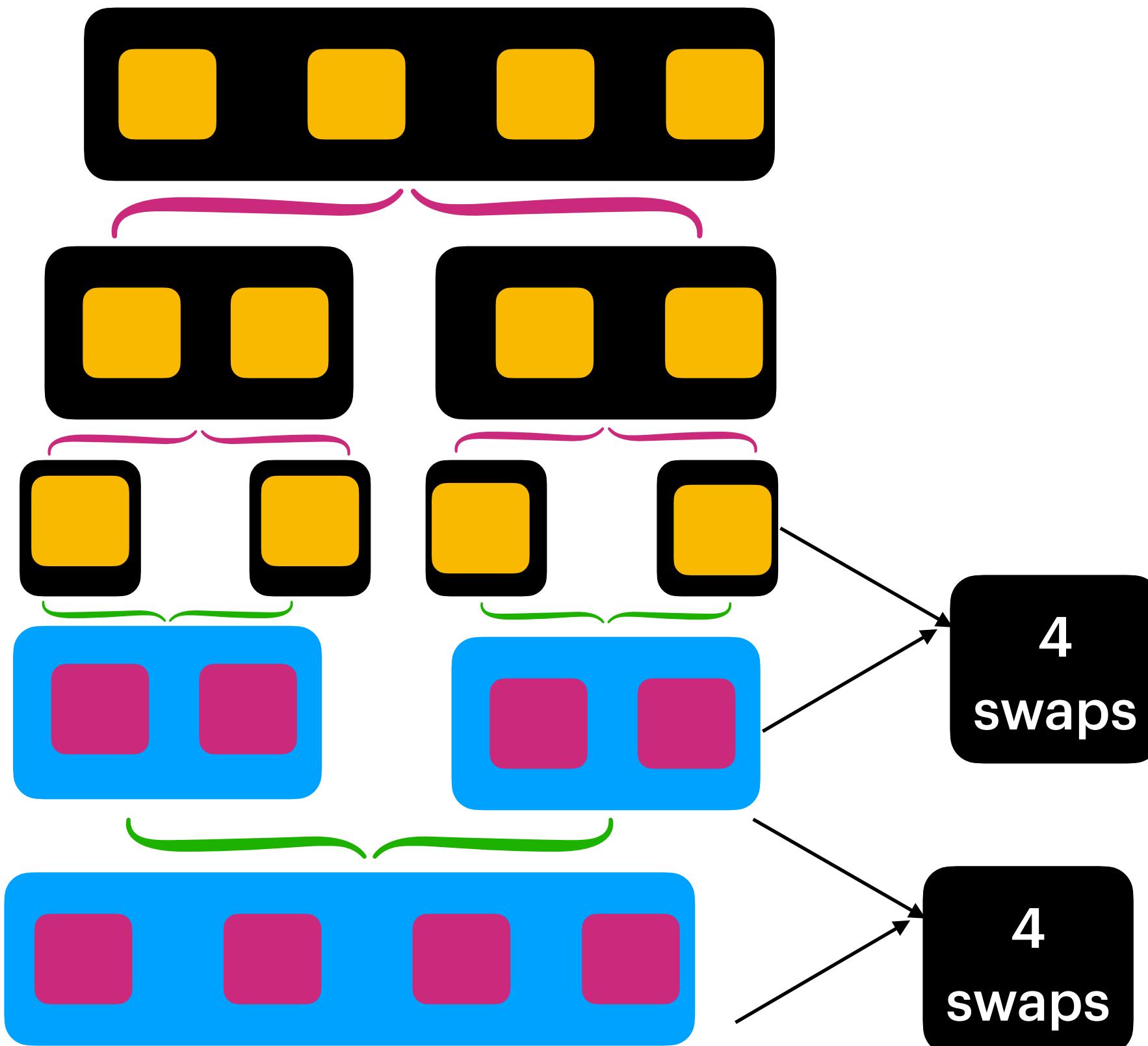
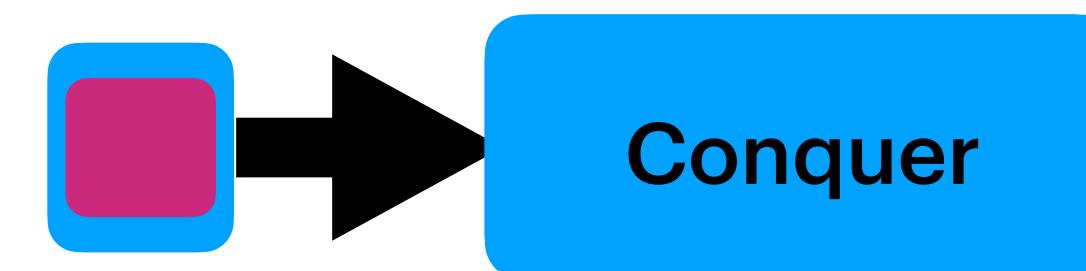
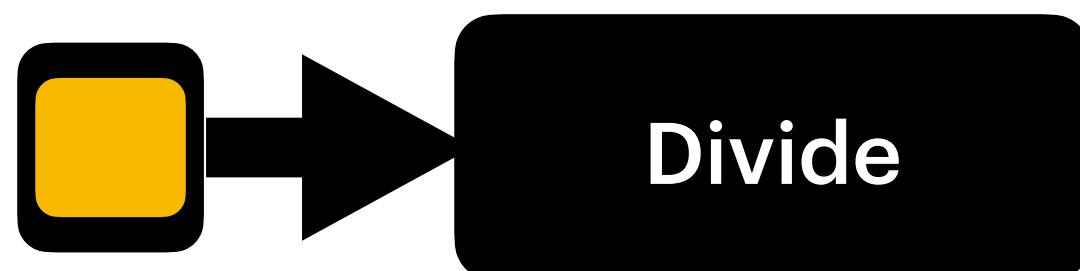
Let's figure out  
the Time  
Complexity.

For a merge sort  
the complex  
operations  
happens while  
swapping  
(Conquer logic).

Let's find out  
solution for time  
complexity by  
considering  
swap count.



Merge Sort taken 24 swaps If the size of the array is 8.  
SizeOf(8) => 8swaps in 3steps =  $8 \times 3 = 24$  swaps



Merge Sort taken 8 swaps If the size of the array is 4.  
 $\text{SizeOf}(4) \Rightarrow 4\text{swaps in } 2\text{steps} = 4*2 = 8 \text{ swaps}$

## MergeSort :

For Size Of ( 2 ) = 2 swaps =>  $2 * 1 = 2 * \log_2^1$

For Size Of ( 4 ) = 8 swaps =>  $4 * 2 = 4 * \log_2^2$

For Size Of ( 8 ) = 24 swaps =>  $8 * 3 = 8 * \log_2^3$

Finally for a Merge Sort we can derive a time complexity as  $n \log n$ .

Time Complexity =  $O(n \log n)$

Space Complexity =  $O(n)$

Recursive / Non Recursive = Recursive

Stability = Stable

{4,1,4,3}

{4,1} {4,3}

{4} {1} {4} {3}

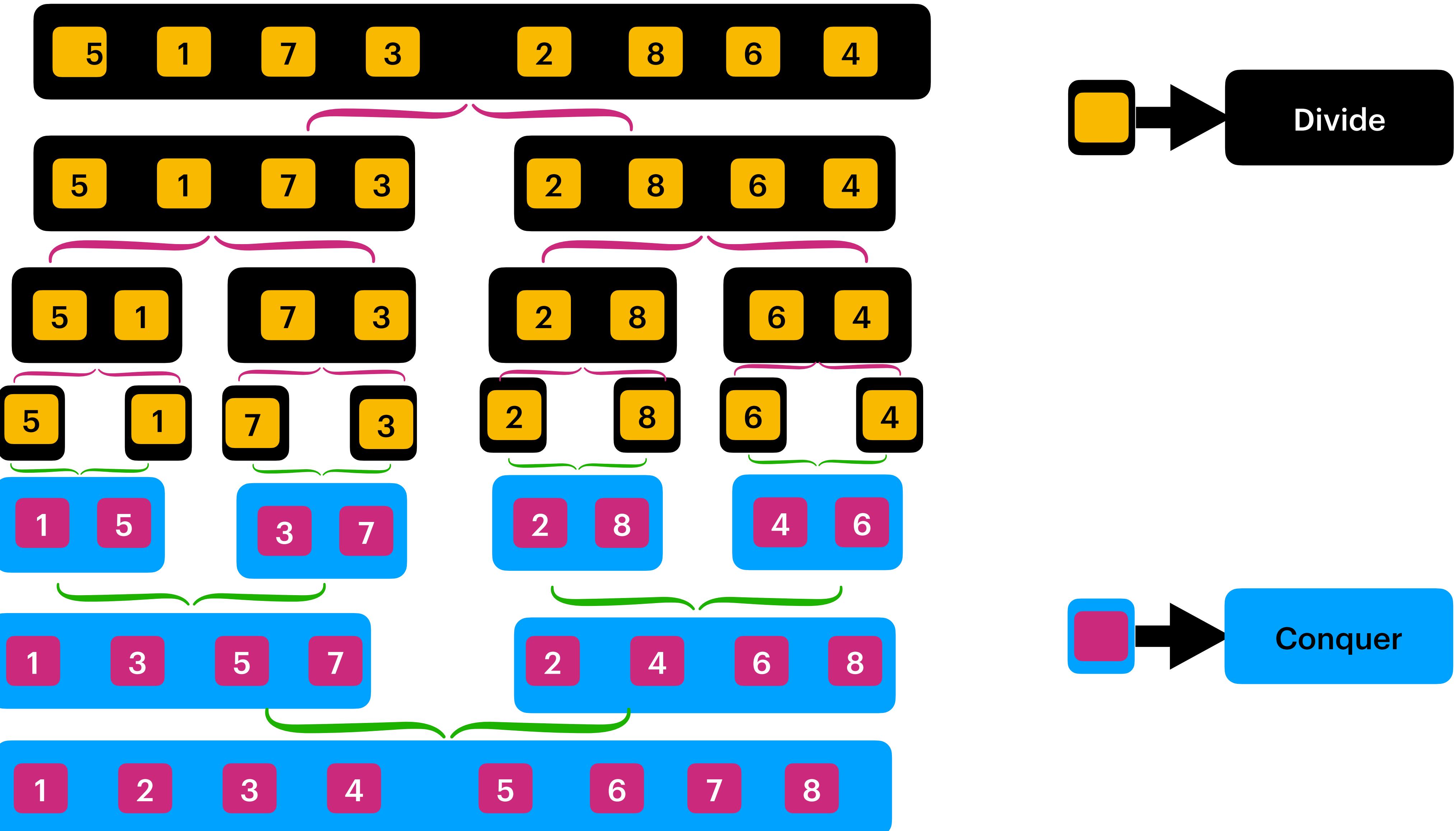
{1,4} {3,4}

{1,3,4,4}

Internal / External Sort = Can be used for both.

Comparison Sort = Yes

Swap =  $O(n \log n)$



First we apply divide and conquer on left array then on right array recursively.

This occurs recursively  
So that at any given point in time  
the number of active StackFrame  
count is  $\log n$

The flow is based on StepNumber please observe .

mergeSort(arr)

divide(arr)  $n = 4$   
 $\{5,1,7,3\}$

Step6 : leftArr output  $\{1,5\}$   
rightArr output  $\{3,7\}$   
Here conquer at  $n=4$   
final out put  $\{1,3,5,7\}$

Step4 : As divide of leftArr & rightArr completed we do the conquer .i.e  $\{1,5\}$

divide(leftArr)  $n = 2$   
 $\{5,1\}$

divide(rightArr)  $n = 2$   
 $\{7,3\}$

Step5 : now recursion start at rightArr. Process is exactly same,  
(follow leftArr recursion)  
final output  $\{3,7\}$

divide(leftArr)  $n=1$   
 $\{5\}$

Step1 : StackFrame Count : 1

Represents divide logic

~~divide(leftArr)  $n=1$~~

Step2 :  $n == 1$  this stackframe will terminate

~~divide(rightArr)  $n=1$~~

Step3 : As  $n == 1$  this stackframe will terminate

Represent conquer logic

## Finally About Merge Sort :

Merge Sort follows the divide & conquer pattern. Which divides array into possible sub problems then do the conquering .

As the merge sort does the sorting using Out-Place algorithm it takes  $O(n)$  Space complexity.

MergeSort is good for External Sorting.

Time Complexity =  $O(n \log n)$

Space Complexity =  $O(n)$

Recursive / Non Recursive = Recursive

Stability = Stable

{4,1,4,3}  
{4,1} {4,3}  
{4} {1} {4} {3}  
{1,4} {3,4}  
{1,3,4,4}

Internal /Eternal Sort = Can be used for both.

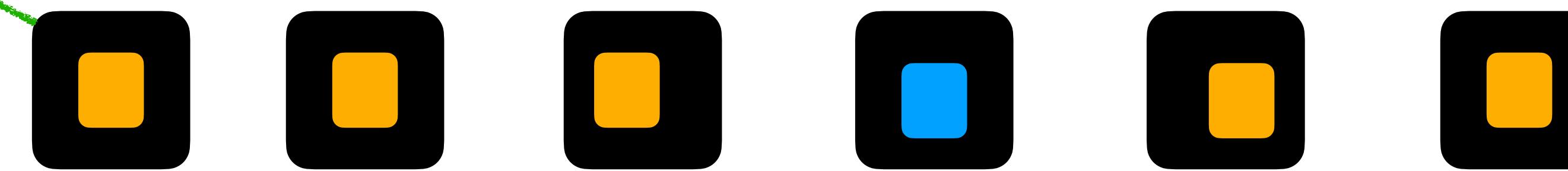
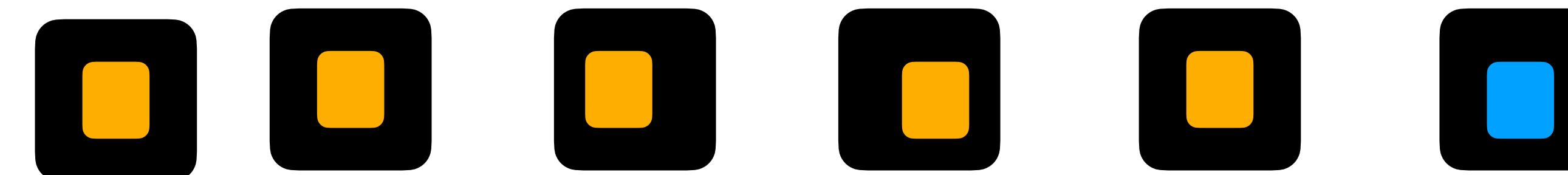
Comparison Sort = Yes

Swap =  $O(n \log n)$

**Quick Sort :** Quick Sort uses the divide & conquer approach. Quick Sort chooses the pivot element within a collection of elements then do the sorting such that elements less than the pivot moved to left part and the elements greater than pivot moved to right part.

Same logic applies on left Part and Right Part recursively.

Elements are sorted with respect to pivot. This little observation makes QuickSort to do only log comparisons in recursive calls.

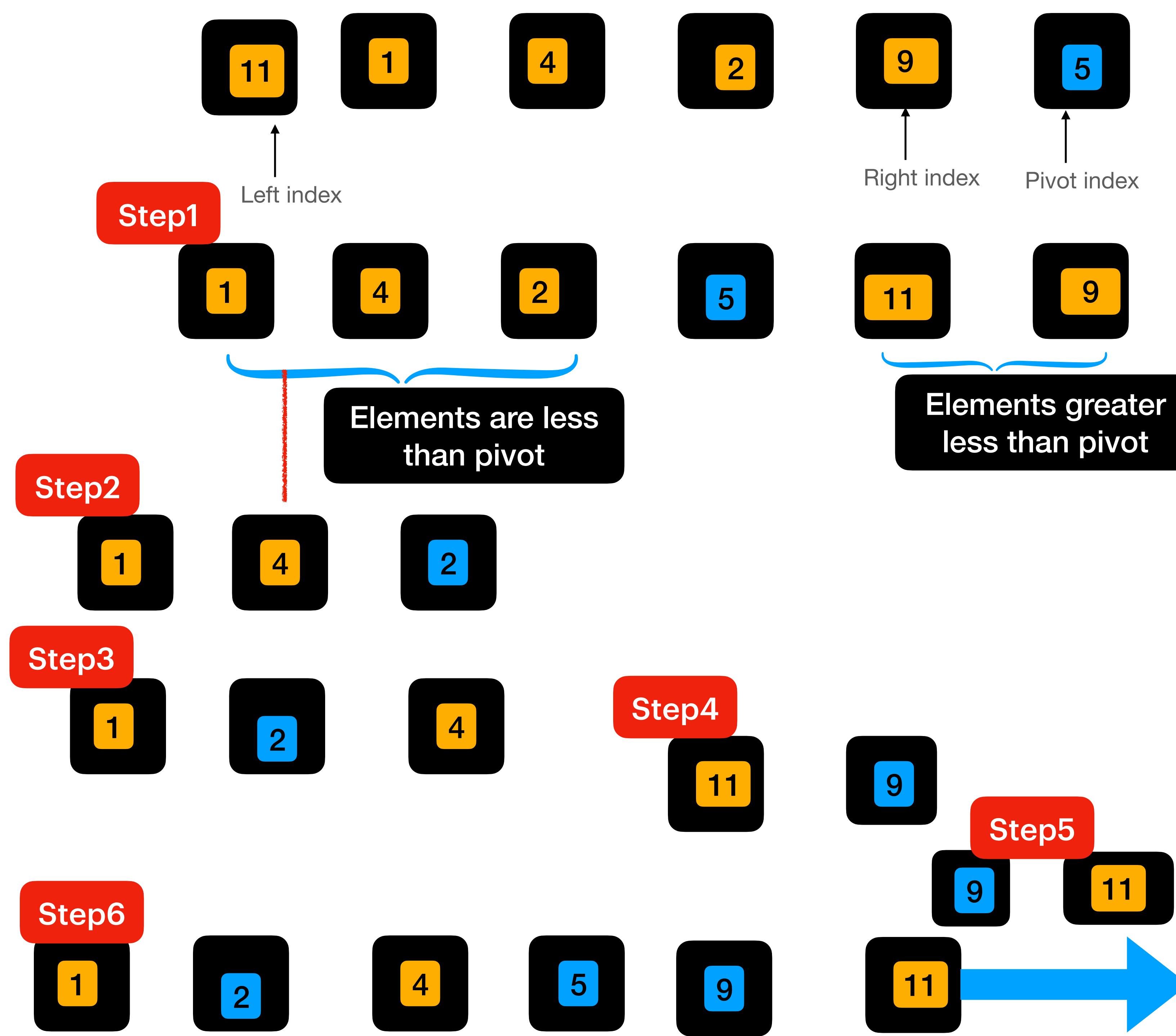


Elements are less than pivot

Elements greater than pivot

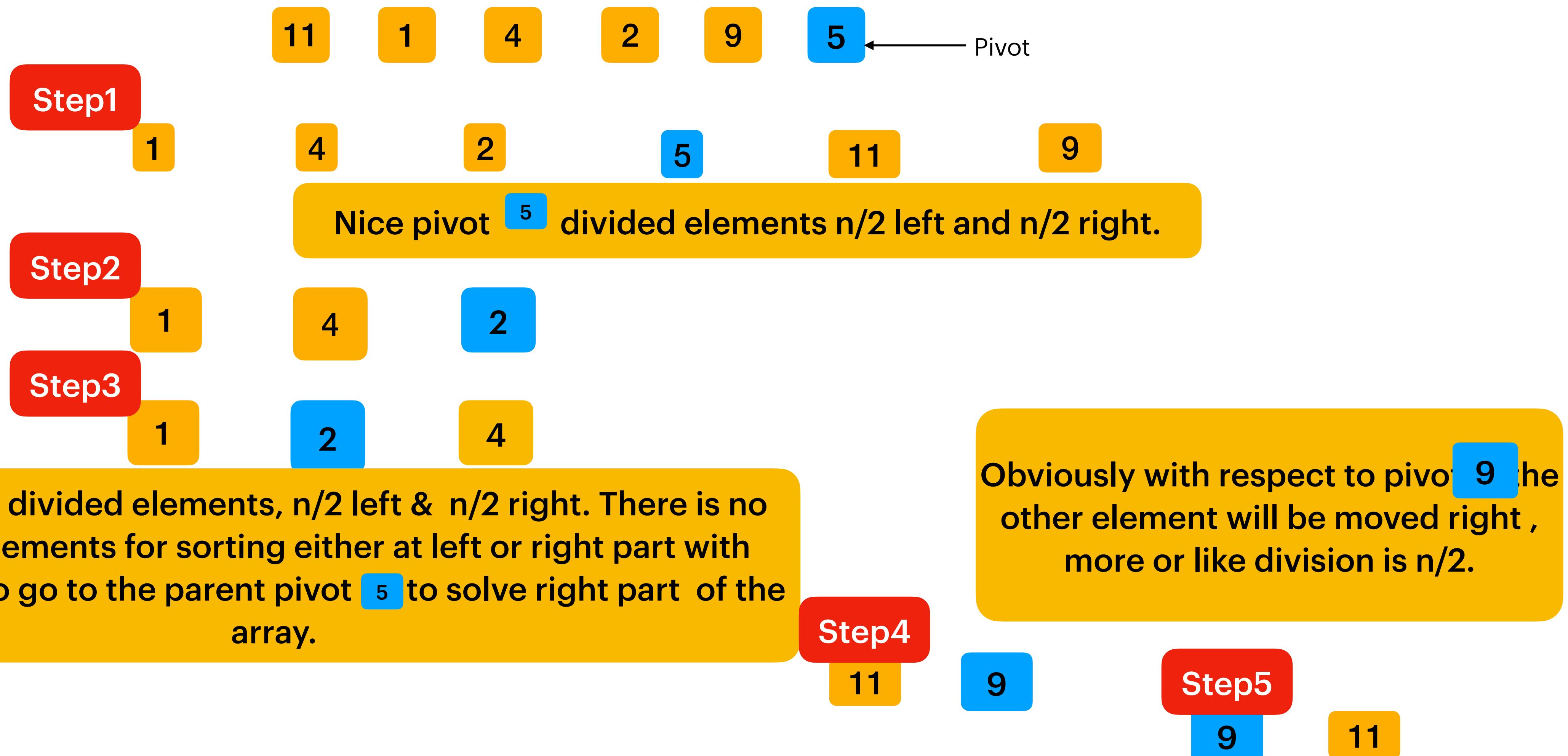


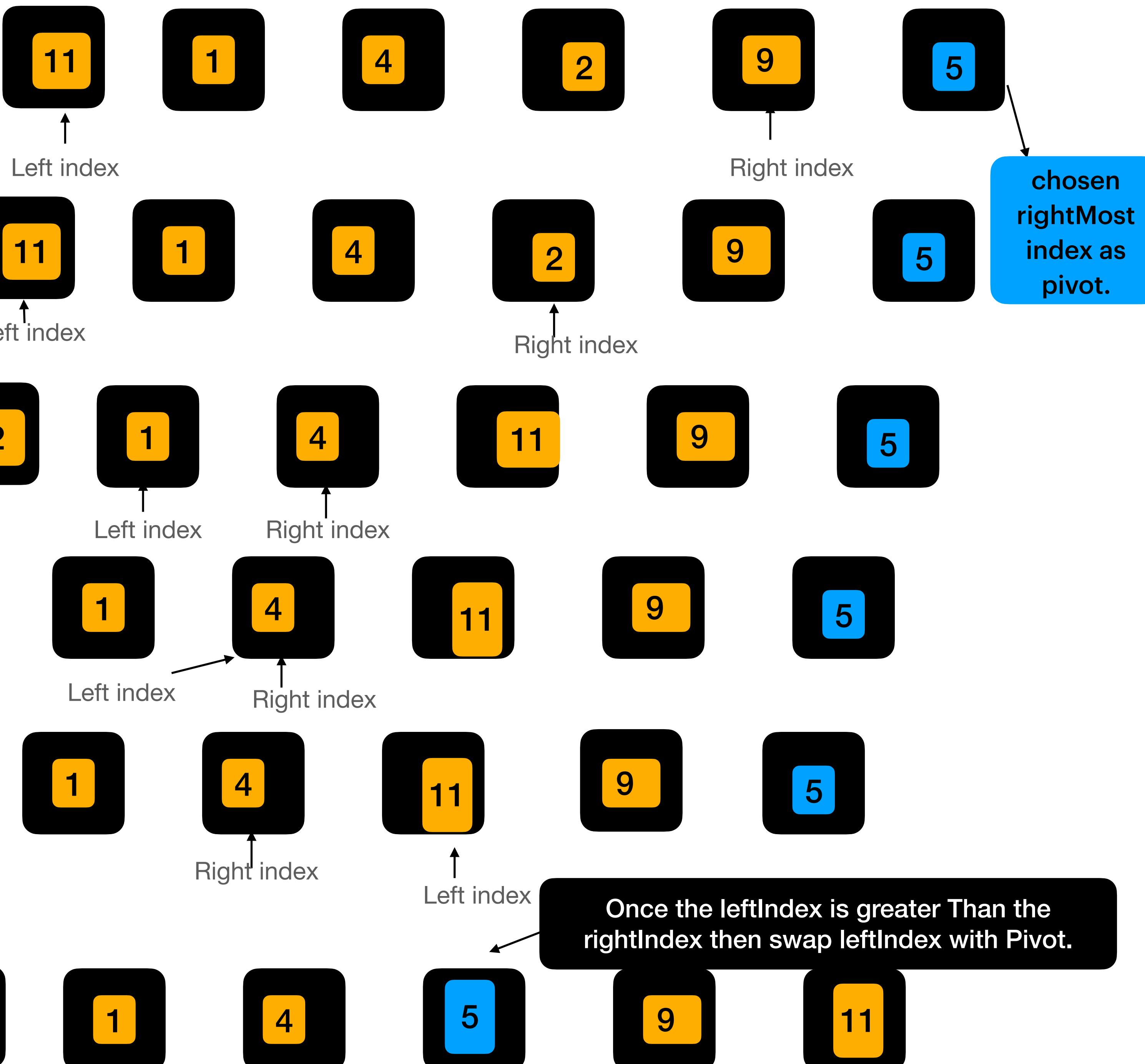
In Quick Sort choosing a pivot is a critical part. In each recursive step the pivot has to partition the elements half (i.e  $n/2$  &  $n/2$  for both left and right) . So that in the next recursive call either in leftPart or in rightPart takes only log comparison.



Choosing Pivot in Quick sort makes a critical role.  
Let's consider for simplicity we chooses rightMost Index as Pivot.

Assume input elements are in unsorted order





=> Quick sort uses InPlace Algorithm :  
 This is the advantage over the merge sort.  
 QuickSort takes O(1) Space complexity for sort the elements.

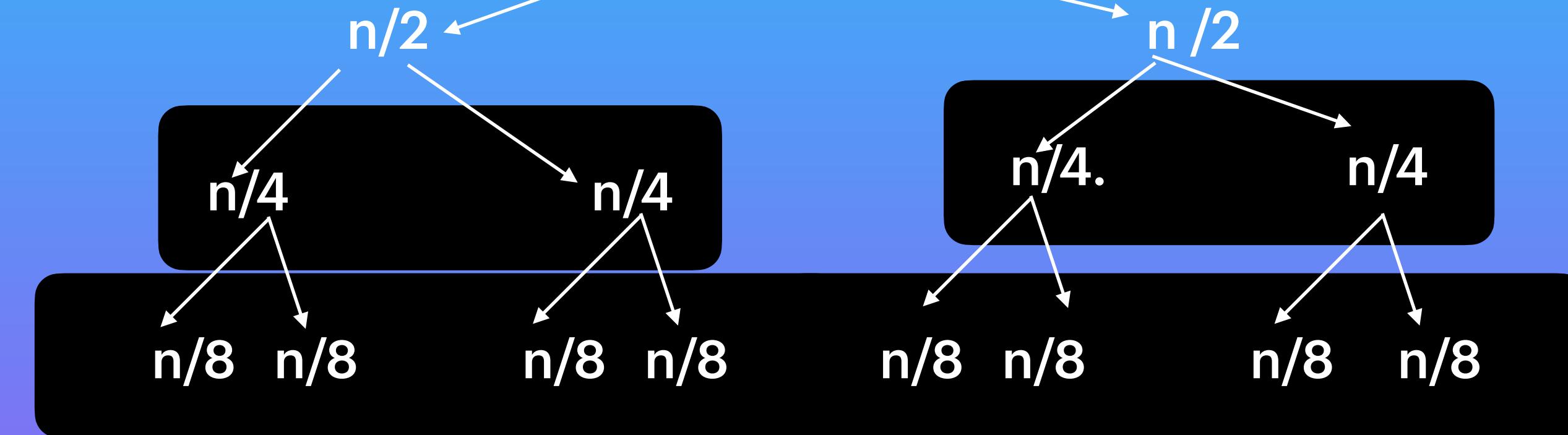
```
while (leftIndex <= rightIndex ) {
```

1. Make sure leftIndex points to lower index.
2. Make sure rightIndex points to higher index (pivotIndex - 1).
3. If the leftIndex value less than pivot then move leftIndex to right.
4. If the rightIndex value greater than pivot move rightIndex to left.
5. When the leftIndex value > pivot && rightIndex value < pivot then swap  
 $\text{leftIndex}++$   
 $\text{rightIndex}--$



Let's figure out the TimeComplexity for QuickSort. If the elements are unsorted.

n {11,1,4,2,9,5,20,15,12} unsorted Array



Now when the elements are unsorted  $\{11,1,4,2,9,5\}$  in a quick sort either in best or average Case => In each level pivot divides into  $n/2$  left and  $n/2$  right sub problems . In each sub problem altogether sort happens at left and right recursively.

$$\begin{aligned} \text{As per diagram } n &= (\text{leftPart Recursion}) + (\text{rightPart Recursion}) \\ &= n/2 * \log n + n/2 * \log n = 2 * n/2 * \log n = n \log n \end{aligned}$$

For  $n$  pivot values  $\log n$  comparisons happens at each level so that Time Complexity is  $O(n \log n)$

Let's figure out the TimeComplexity for QuickSort. If the elements are sorted.



n {1,5,6,8,10} Sorted Array

n-1  
0

n-2  
0

n-3  
0

n-4  
0

Now when the elements are sorted {1,5,6,8,10}, If we consider always rightMost Index as pivot then in each level pivot divides into (n-1) left and 0 right sub problems .

In each sub problem sort happens only at left part recursively.

As per diagram  $n = (\text{leftPart Recursion}) + 0 = n(n-1) + n(n-2) + \dots + 1 = n^2$

For a sorted array QuickSort gives  $O(n^2)$  Time Complexity.

This can be addressed by taking proper pivot.

## Pivot Techniques :

Usually for simplicity in a quick sort either we take leftMost or rightMost index a pivot but it leads to  $O(n^2)$  time complexity when the elements in an array are sorted so that choosing right pivot is always makes your algorithm best in QuickSort.

### Here are the Pivot Techniques :

1. Always choose random index as a pivot index, it results into  $O(n \log n)$  time complexity.  
(This technique is been used in our code example refer QuickSortApp.java)

2. If you are a good mathematician you can take a few elements in a given array as a sample, identify the behaviour of elements then choose the pivot .

( OR )

you can take last few elements find the median then choose the pivot.

3. You can also have dual pivots in your algorithm to improve the performance of a quick sort.

## When to go for QuickSort ?

If most of the elements are already sorted then go with InsertionSort, which gives you best time complexity i.e  $O(n)$ .

If the elements are Unsorted then you can prefer Quicksort with proper pivot technique, which gives you best time complexity i.e  $O(n \log n)$ .



## Finally On QuickSort

TimeComplexity :

**Best Case or Average Case =  $O(n \log n)$**

**Worst Case =  $O(n^2)$**

$n^2$  would be avoided if you go with proper pivot even in worst case.

Space Complexity :

As its following InPlace algorithm so that Space Complexity is  $O(1)$

If you are more particular about active stack frames for quick sort Space Complexity would be  $O(\log n)$

**Stable : Unstable**

**Recursive / NonRecursive => Recursive**

**Internal / External => Internal Sort**

**Comparison => Yes**

**Swaps =>  $\log n$  ( As we go with proper pivot)**

**How Come QuickSort gives best performance when both MergeSort & QuickSort gives same Time Complexity i.e  $O(n\log n)$  ?**

Actually quick sort uses InPlace algorithm so the sort happens within the array. In CPU there is branch prediction feature , in simple branch prediction is nothing but maintaining the cache & guessing the next requested value.

This is what makes the difference between QuickSort and MergeSort.

As QuickSort performs sorting within the array(InPlace Algorithm) , the CPU branch prediction feature helps to execute faster.

Where as MergeSort uses out place algorithm, it results into effecting execution time.

Merge Sort work efficiently in external sorting. Always use QuickSort for internal Sorting , use MergeSort for external sorting.

## Know more about Recursion :

As we know that for every Recursive Call separate StackFrame would be created.

Working with Recursion is a costly job, as its going to occupy more space and also there is a limit for StackFrames .

Unhandily way of Recursion could cause StackOverFlow.

To avoid this we should use Recursion only when it is needed.

Thumb rule for a Recursion Problem is , for a Recursive problem , we always take a decision based on choice.  
Here choice is either include or exclude.

For Ex : given input "ab" find out all the possible subsets.

Output : "", "a", "b", "ab"

1. For subset "" empty String we excluded 'a' & 'b'
2. For subset "a" we include 'a' & exclude 'b'
3. For subset "b" we excluded 'a' & included 'b'
4. For subset "ab" we include 'a' & 'b'

If you take a look at output



output	a	b
""	✗	✗
a	✓	✗
b	✗	✓
ab	✓	✓

## Steps to work with Recursion :

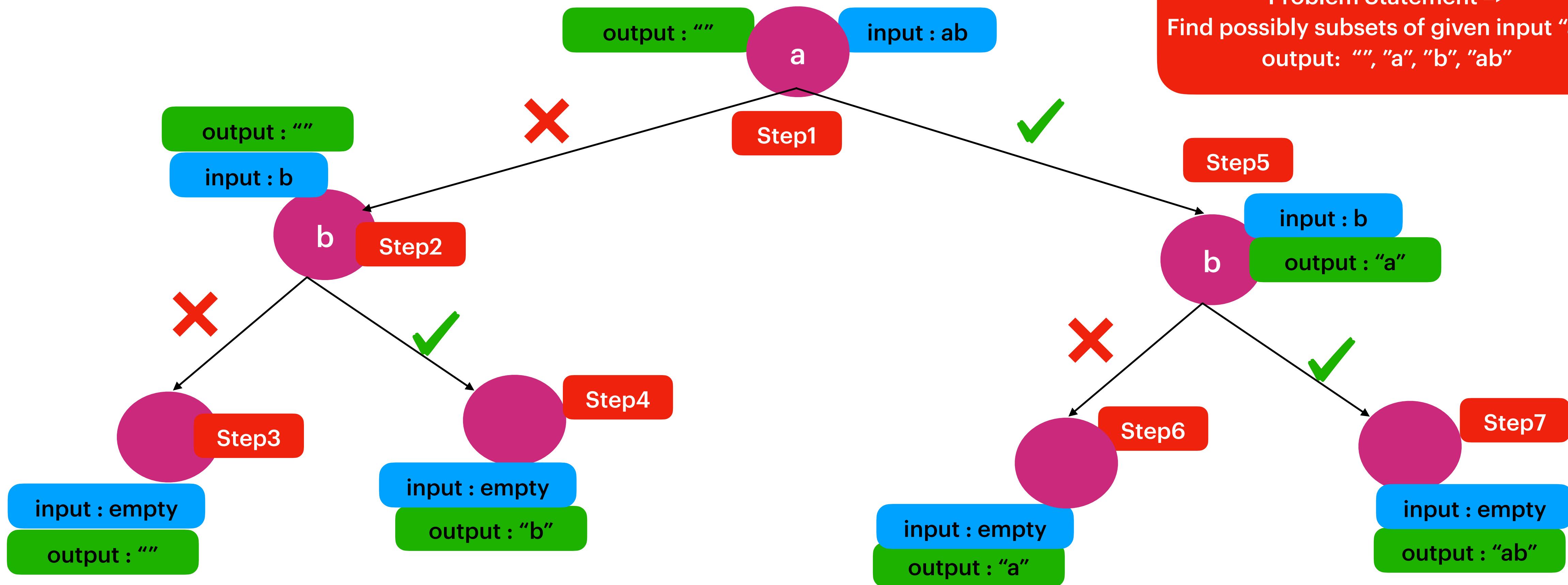
**When we are working with Recursion first think about the base condition.**

**Base Condition => Always be the smallest possible Valid value for a given use case.**

**Recursion Tree => Draw a Recursion Tree , for every sub problem you will find two possible nodes , one is with exclude and other one is with include.**

**Write the Code => We can solve any Recursion problem with two lines of code.**

**Problem Statement =>**  
Find possibly subsets of given input "ab".  
output: "", "a", "b", "ab"



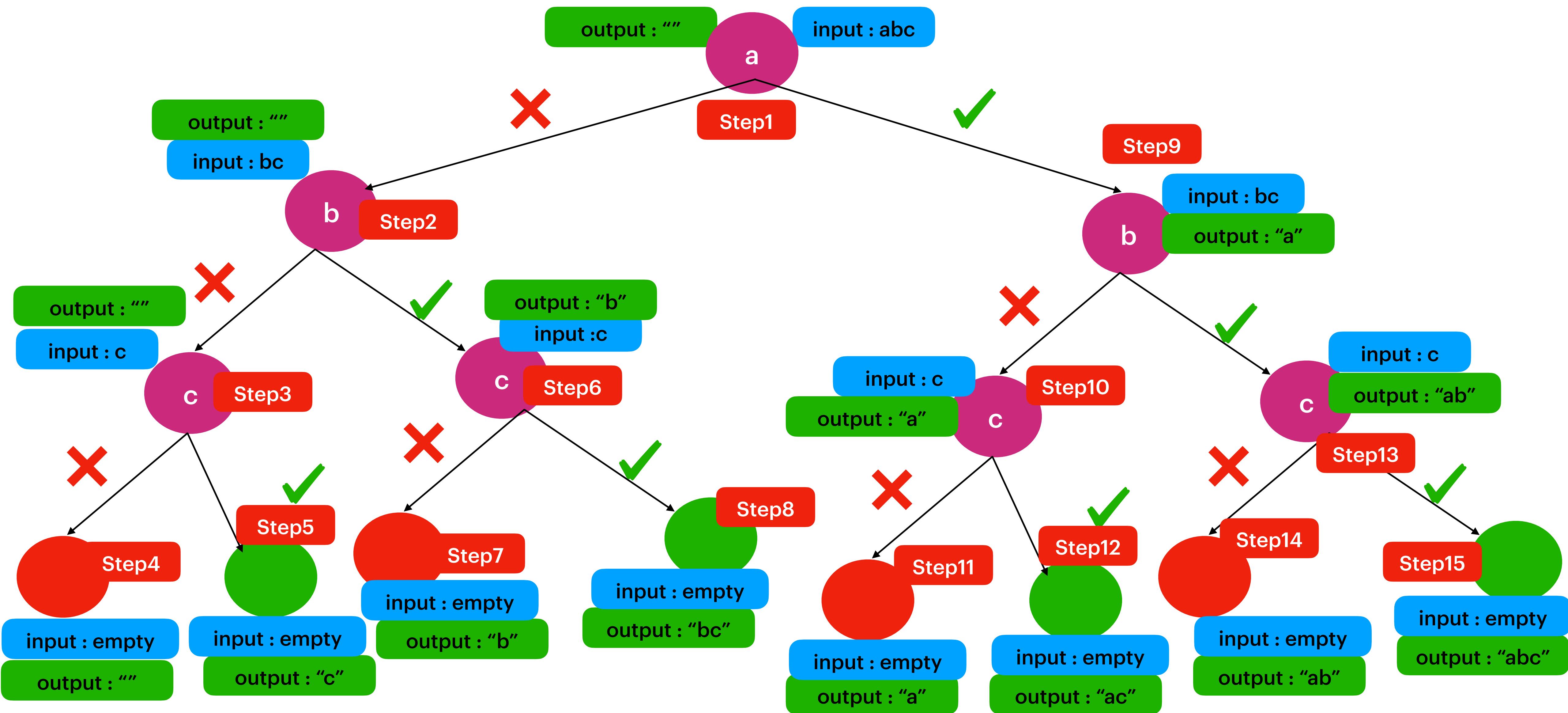
Time Complexity :  $2^n$   
Space Complexity :  $O(n)$

```

public void subSets(String input, String output)
{
    if(input.isEmpty())
    {
        System.out.println(output);
        return;
    }

    //Exclude
    subSets(input.substring(1), output);

    //Include
    subSets( input.substring(1), output+input.charAt(0));
}
  
```



Sorted array has binary search tree property. If you take a "mid" element, elements less than "mid" would be on left side & elements greater than "mid" would be on right side. This key we can use to divide array to half in every iteration/recursive call.

{1,2,3,4,**5**,6,7,8,9}. => find key = 1

As the mid element **5** > key i.e (1)

Take left part of array {1,2,3,4}. Exclude right side part {6,7,8,9}

As the mid element 2 > key i.e (1)

Take left part of array {1}. Exclude right side part {3,4}

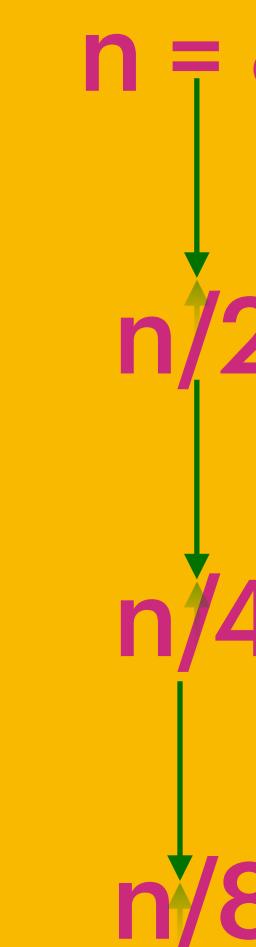
As Mid element 1 == key (i.e 1)

Element found.

If size of the array is 8 => you can find the element in 3 steps

If we use recursion , then we call method recursively logn times. So that there are logn stack frames would be active.

So In case of recursion the SpaceComplexity would be O(logn)



In a binary search for every iteration/ recursion we divide array half. For n = 8 , n/8 is the possible sub problem so this can be reached in 3 steps .

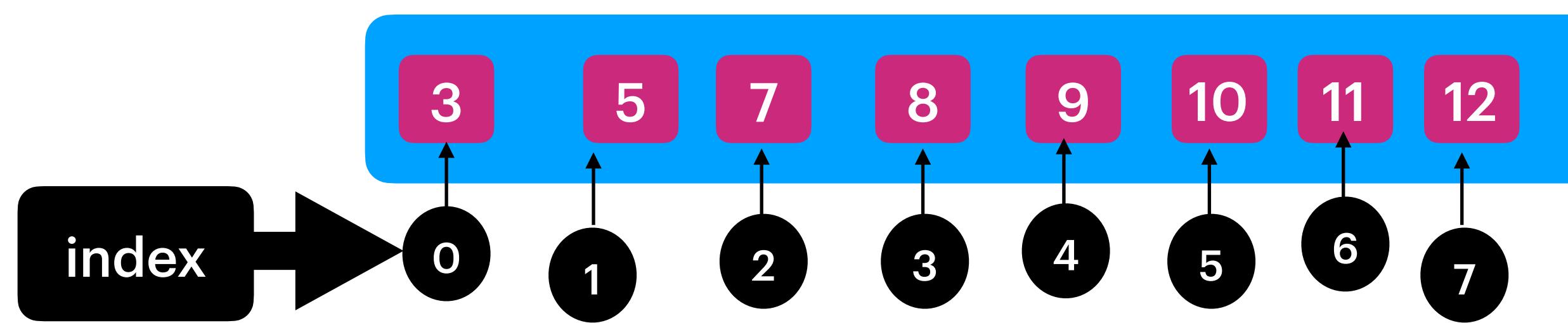
$$\log_2^3 = 8$$

So the TimeComplexity is O(log(n))

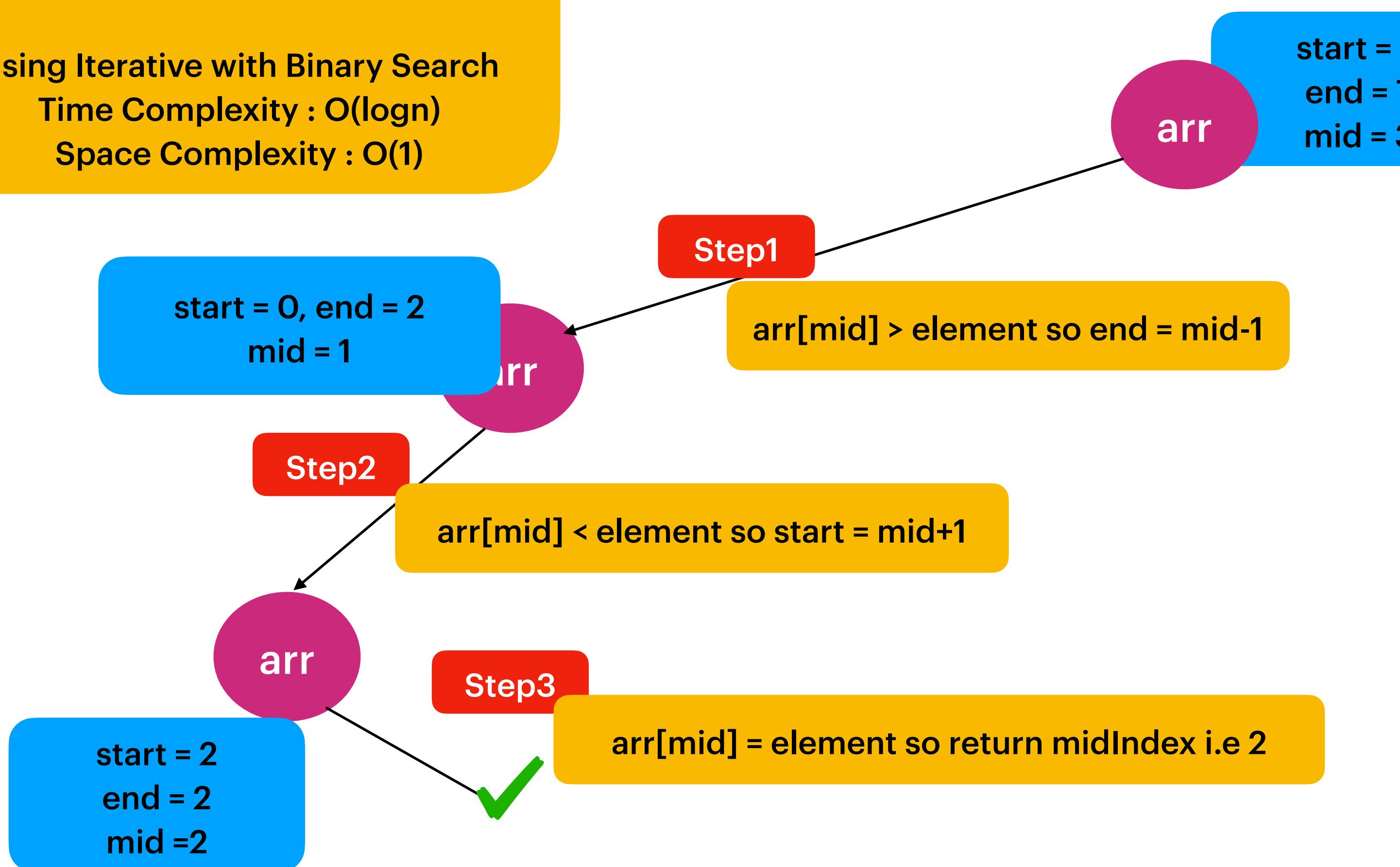
Using iterative  
Time Complexity : O(n)  
Space Complexity : O(1)

Using Recursion with Binary Search  
Time Complexity : O(logn)  
Space Complexity : O(logn)  
As logn stack frames were active

Using Iterative with Binary Search  
Time Complexity : O(logn)  
Space Complexity : O(1)



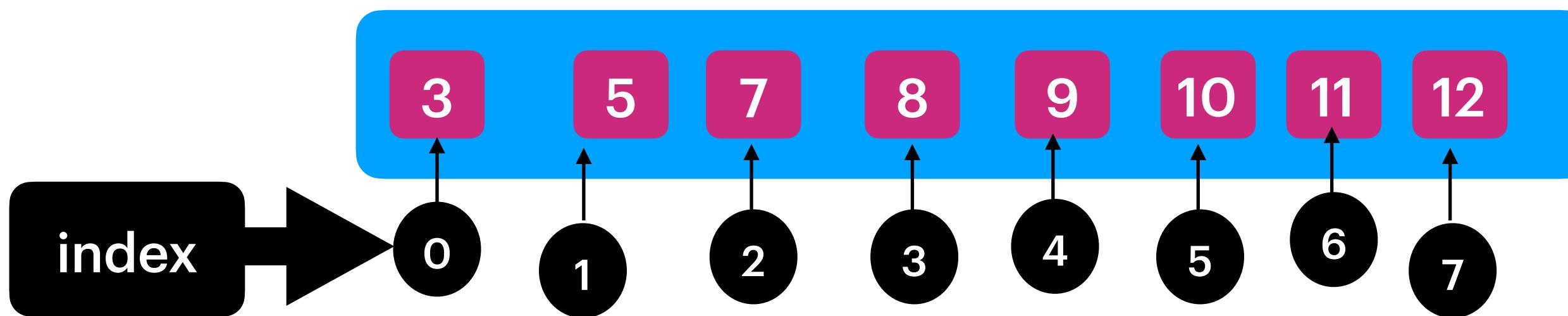
Find Element 7



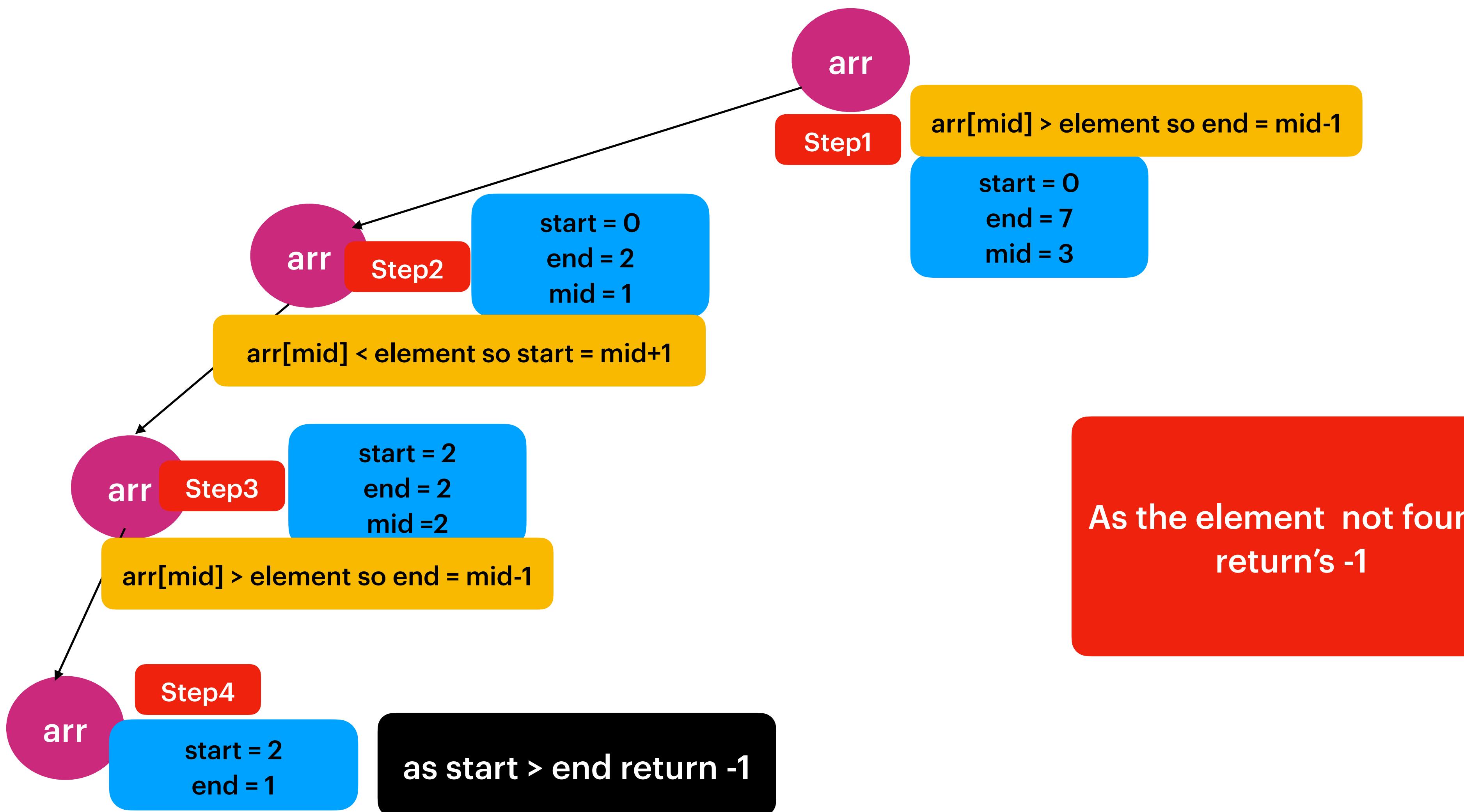
Algorithm to Find an element in  
SortedArray:  
Its Binary Search

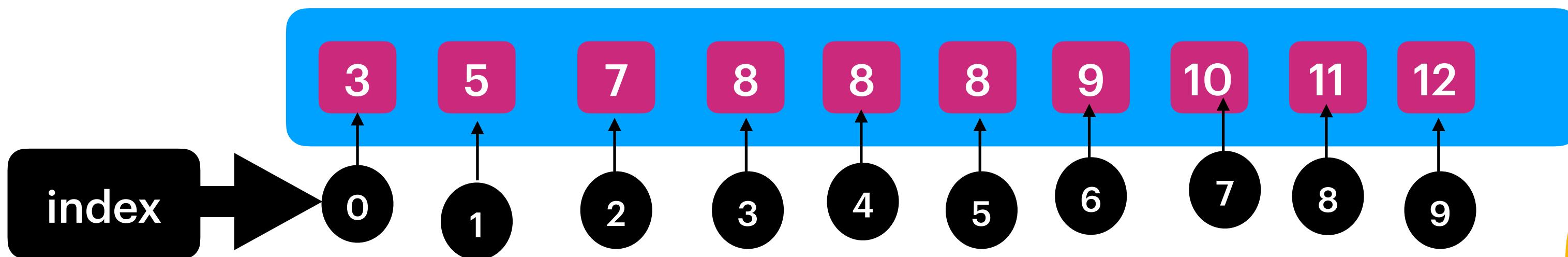
initialise start = 0  
end = arr.length-1

1. Find the mid , if the arr[mid] == element then return mid index.
  2. If the arr[mid] > element then move left side  
end = mid - 1
  3. If the arr[mid] < element then move right .  
i.e start = mid+1
- Base Condition start <= end



Find Element 6





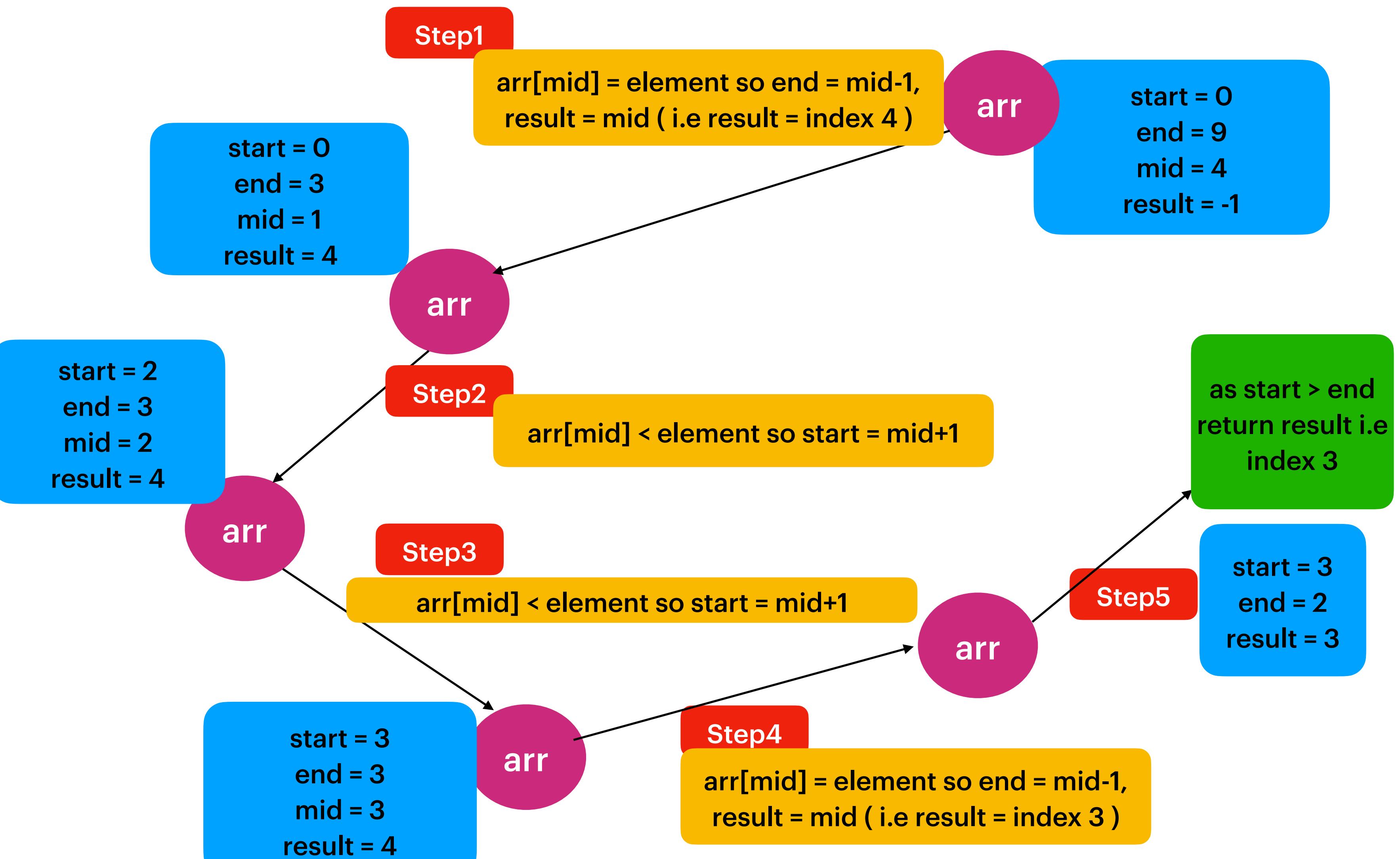
## Find First Occurrence of 8

Algorithm :

Same as Binary Search Algorithm (earlier one)

The only difference is , take a helper variable(i.e result), when we find the element, first update the helper variable then move to left side of the array to reach first occurrence.

when  $\text{arr}[\text{mid}] == \text{element}$   
 $\text{result} = \text{mid}$   
 $\text{end} = \text{mid} - 1$



Using iterative

Time Complexity :  $O(n)$   
Space Complexity :  $O(1)$

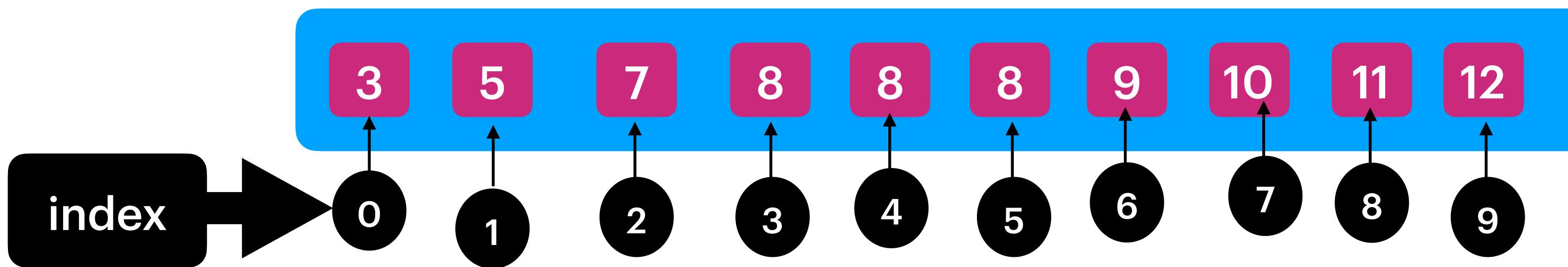
Using Recursion with Binary Search

Time Complexity :  $O(\log n)$   
Space Complexity :  $O(\log n)$

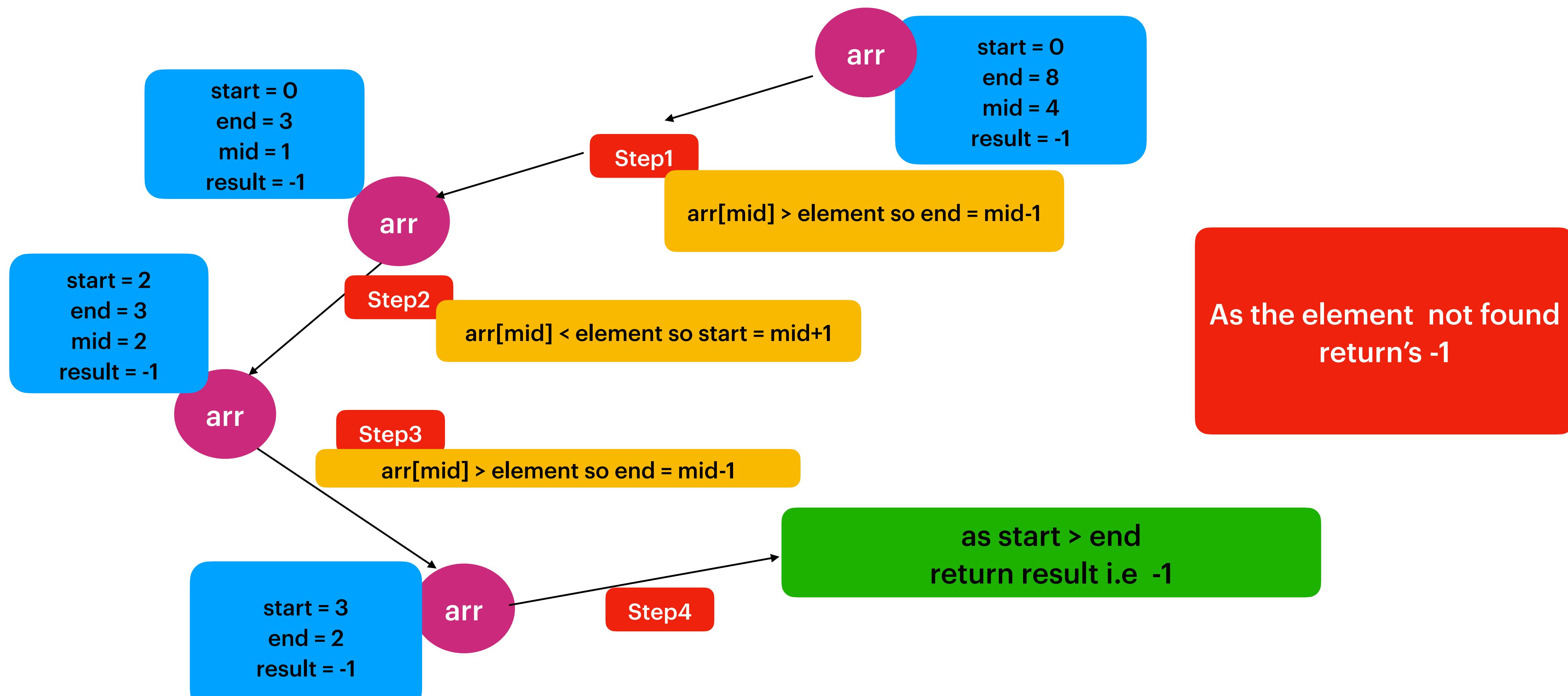
As logn stack frames were active

Using Iterative with Binary Search

Time Complexity :  $O(\log n)$   
Space Complexity :  $O(1)$

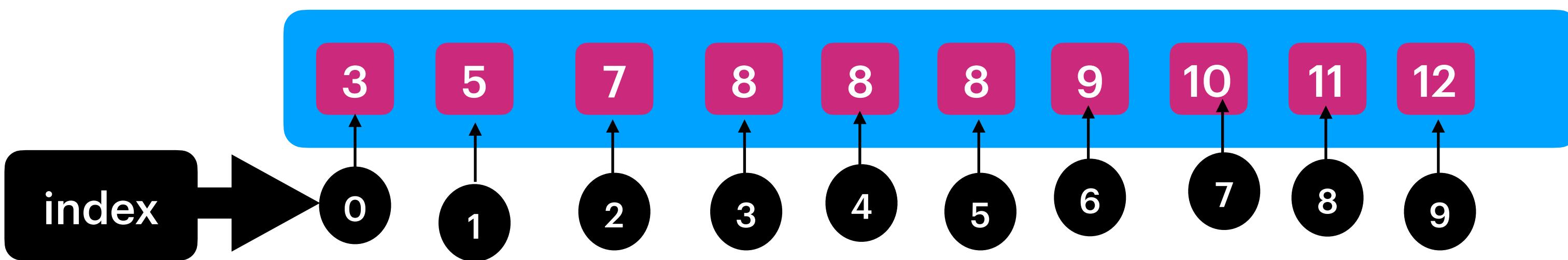


Find First Occurrence of element 6



## Find Last Occurrence of

8



start = 5  
end = 9  
mid = 7  
result = 4

arr[mid] = element so start = mid+1,  
result = mid ( i.e result = index 4 )

start = 0  
end = 9  
mid = 4  
result = -1

Step1

arr

Step2

start = 5  
end = 6  
mid = 5  
result = 4

arr[mid] > element so end = mid-1

arr

Step3

arr[mid] = element so start = mid+1,  
result = mid ( i.e result = index 5 )

start = 6  
end = 5  
result = 5

arr

Step4

as start > end  
return result i.e index 5

Algorithm :

Same as Binary Search Algorithm (earlier one)

The only difference is , take a helper variable(i.e result), when we find the element, first update the helper variable with midIndex , then move to right side of the array to reach last occurrence.

when arr[mid] == element  
result = mid  
start = mid +1

Using iterative

Time Complexity : O(n)

Space Complexity : O(1)

Using Recursion with Binary Search

Time Complexity : O(logn)

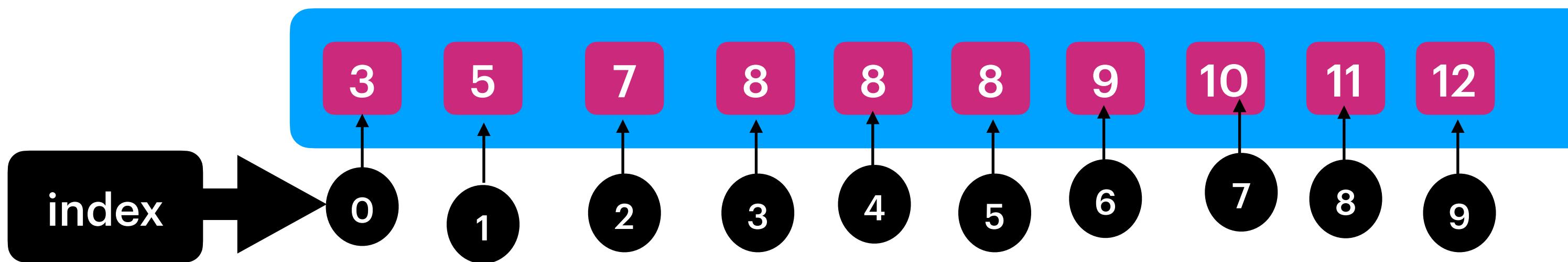
Space Complexity : O(logn)

As logn stack frames were active

Using Iterative with Binary Search

Time Complexity : O(logn)

Space Complexity : O(1)



**Problem Statement : Find Count of element in a sorted array.**

Ex : {3,5,7,8,8,9,10,11,12}

count(8) = 3

count(12) = 1

count(15) = -1

**Algorithm : Using Binary Search**

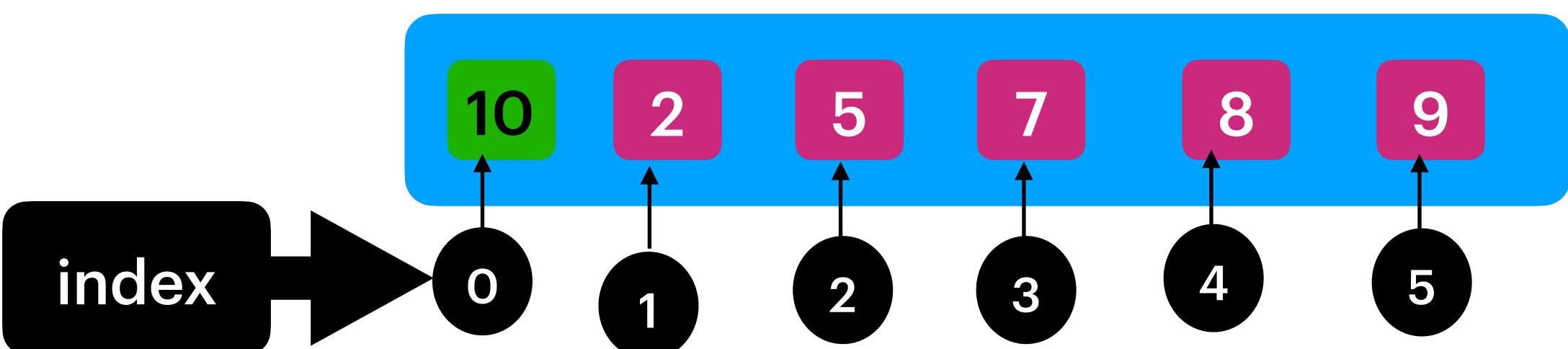
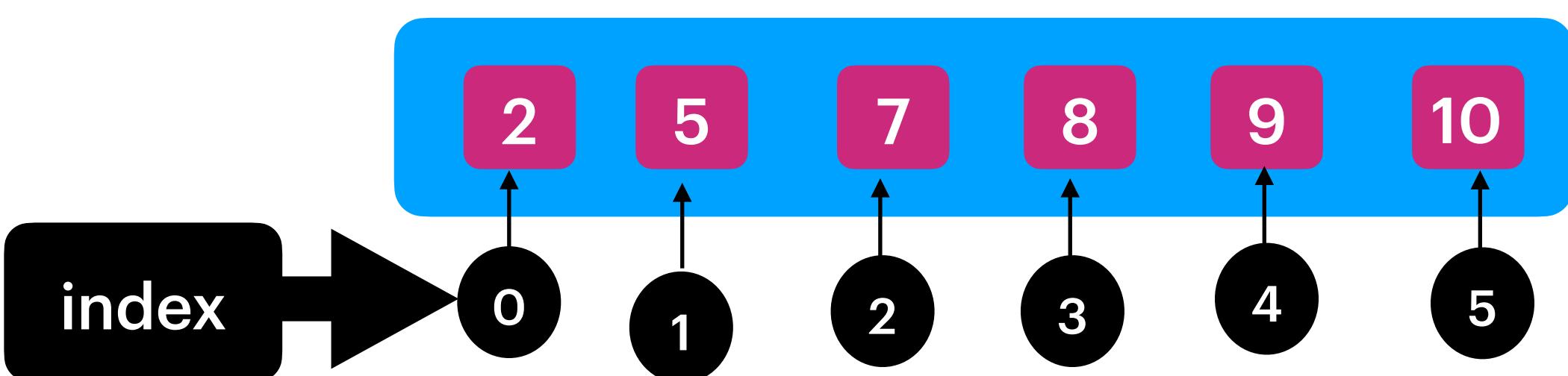
Time Complexity  $O(n)$

Space Complexity  $O(1)$

1. Find out Left Occurrence Index of given element
2. Find out Right Occurrence Index of given element
3. If left and right index's are -1 then return -1  
otherwise return rightOccurIndex - leftOccrIndex + 1

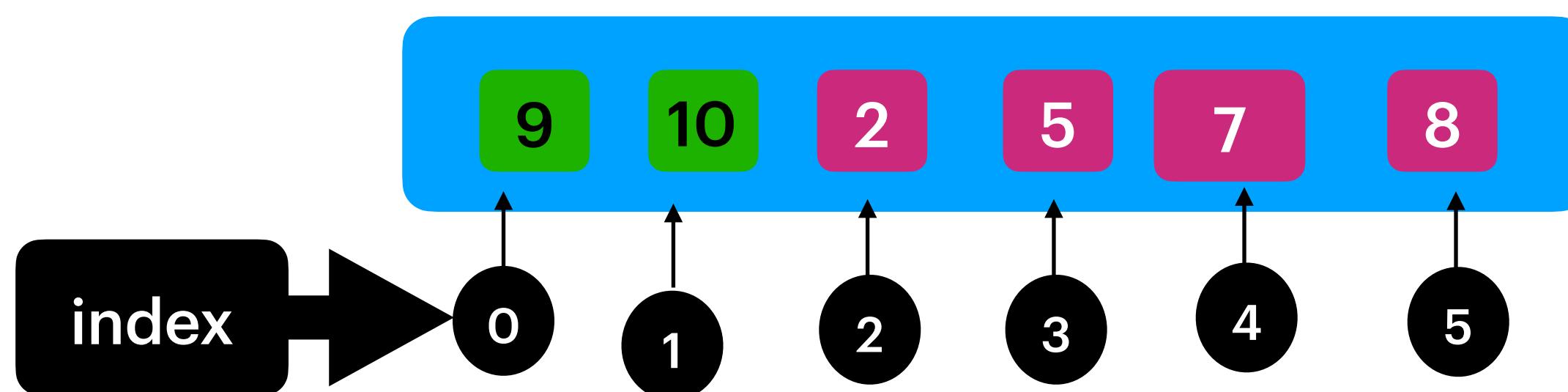
**Problem Statement :**  
Find a RotationCount of a circularly rotated sorted array.

Ex: {3,4,5,1,2} rotation count => 3  
Ex: {1,2,3,4,5} rotation count => 0



Rotation Count 1

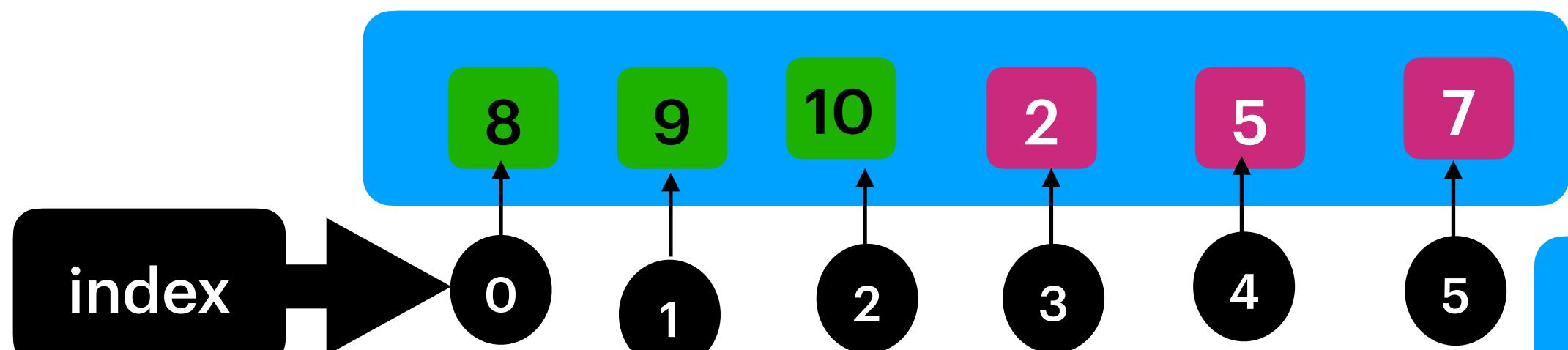
If we closely observe  
rotation count is equals to  
the smallest number index  
within the array.



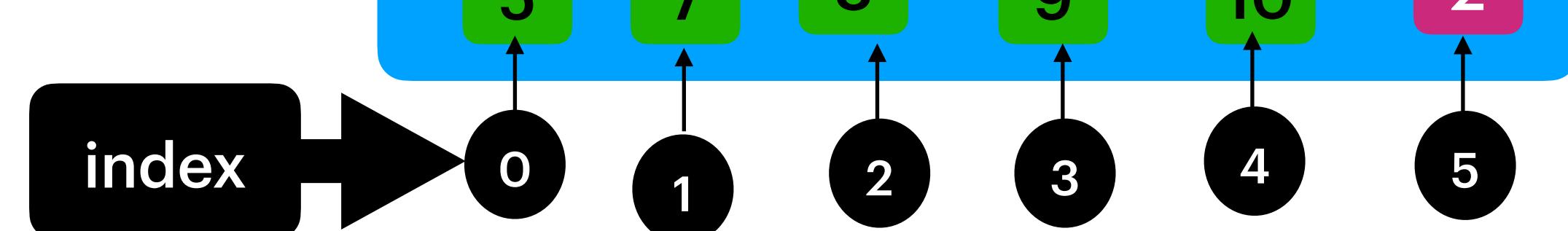
Rotation Count 2

BruteForce  
Approach.

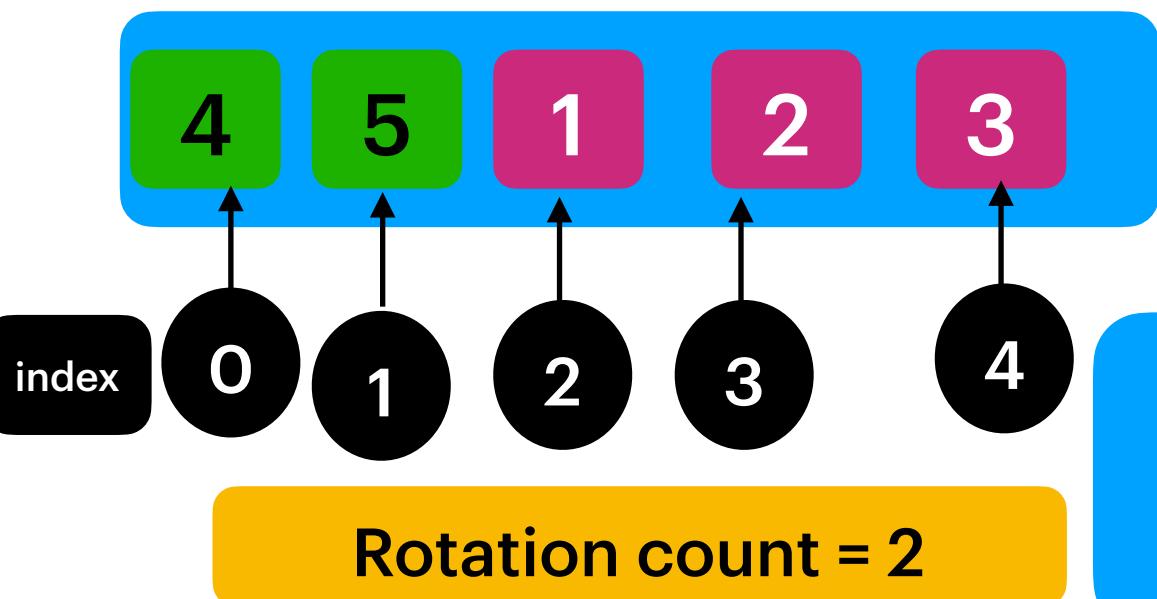
Find the minimum element  
within the array , then return  
the minimum element index.



Rotation Count 3



Rotation Count  
5



Let's track the solution with Binary Search

start = 0  
end = 4  
mid = 2

rotationIndex

Index of minimum element in an array.

As arr[mid-1] greater than arr[mid] so return mid.

return mid i.e 2

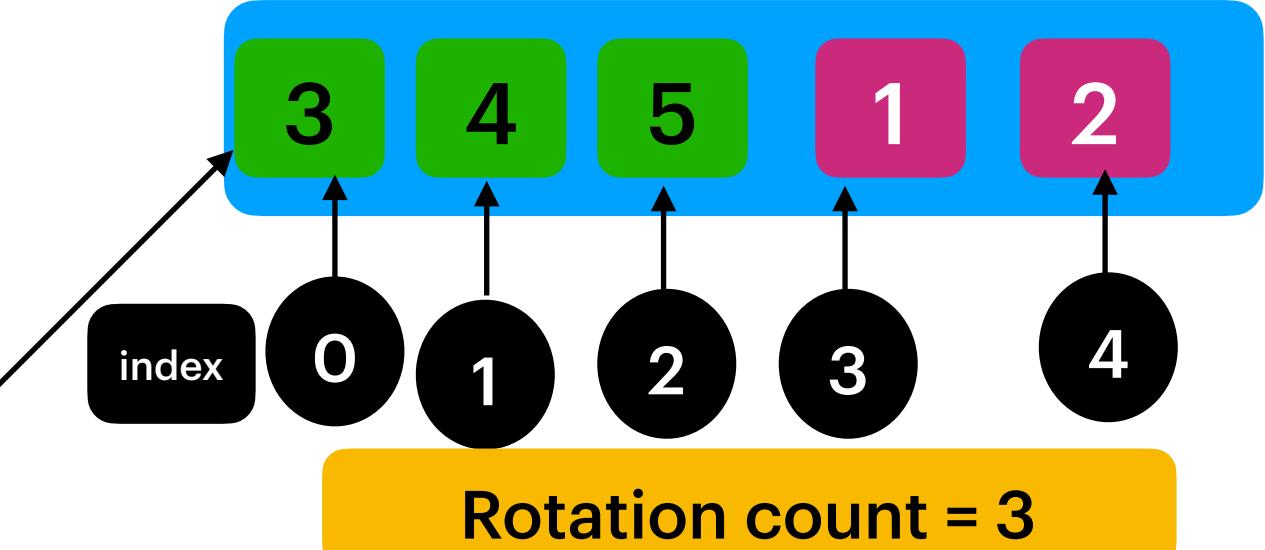
arr

When we work on BinarySearch we always figure out the solution through mid element.

2 possibilities

arr[mid-1] > arr[mid]  
Here "mid" is the rotation index

arr[mid] > arr[mid+1]  
Here "mid+1" is the rotation index

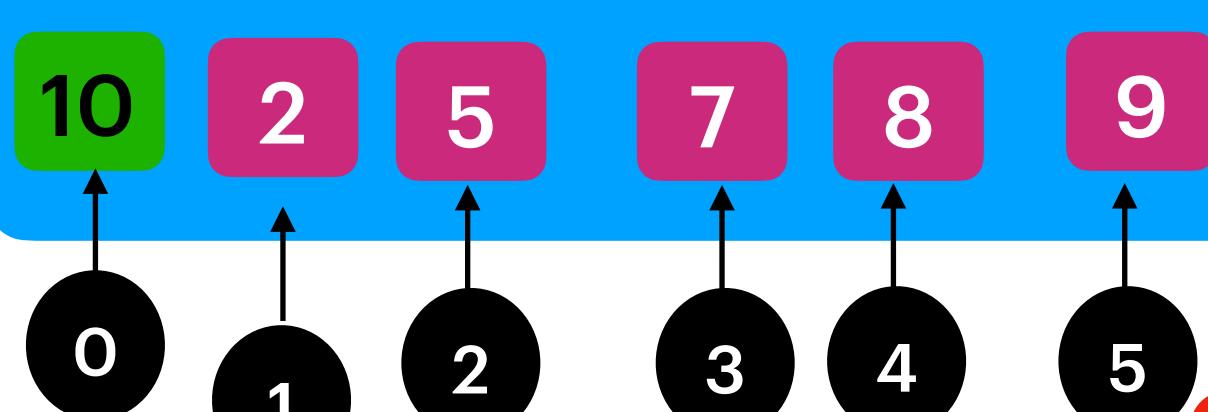


As arr[mid] greater than arr[mid+1] so return mid+1.

return mid i.e 3

arr

start = 0  
end = 5  
mid = 2

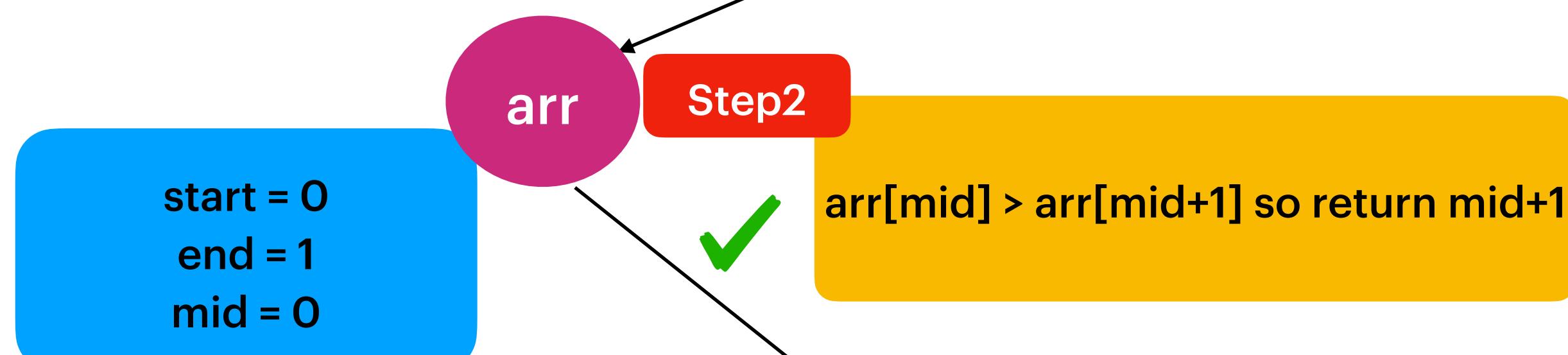
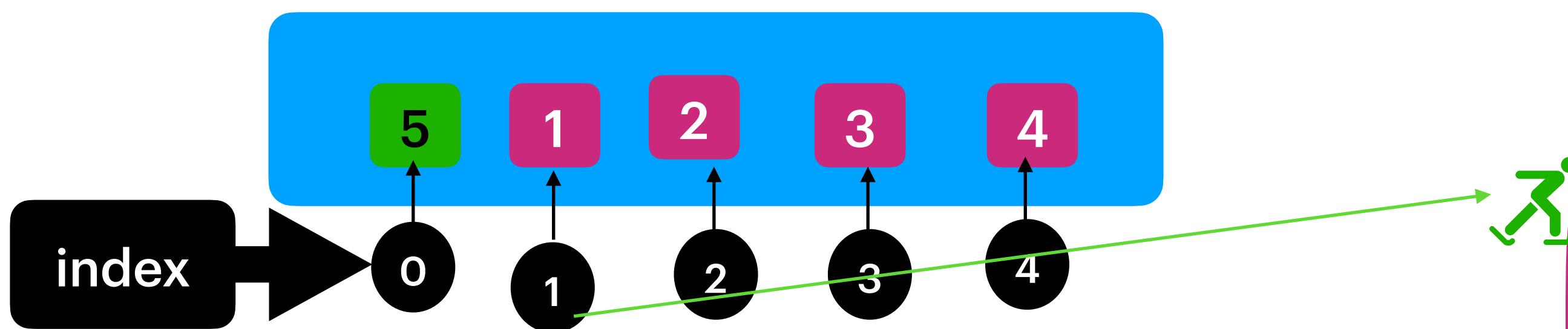


What if you don't find rotationIndex in current iteration. Then we need to think about considering either left part or right part .

2 possibilities

arr[mid] > arr[0] , It means left part is sorted. Move to right  
start = mid + 1;

arr[mid] < arr[end] , It means right part is sorted. Move to left  
end = mid - 1;



Return mid+1 i.e 1

Rotation count = 1

Algorithm to find number of rotation in a circularly sorted array with BinarySearch

1. Initialise start = 0 , end = n-1 , mid = start+ (end-start)/2
2. If the arr[mid] > arr[mid+1] then return mid+1
3. If the arr[mid-1] > arr[mid] then return mid
4. If arr[mid] > arr[0] it means left array is sorted so move to right

i.e start = mid + 1;

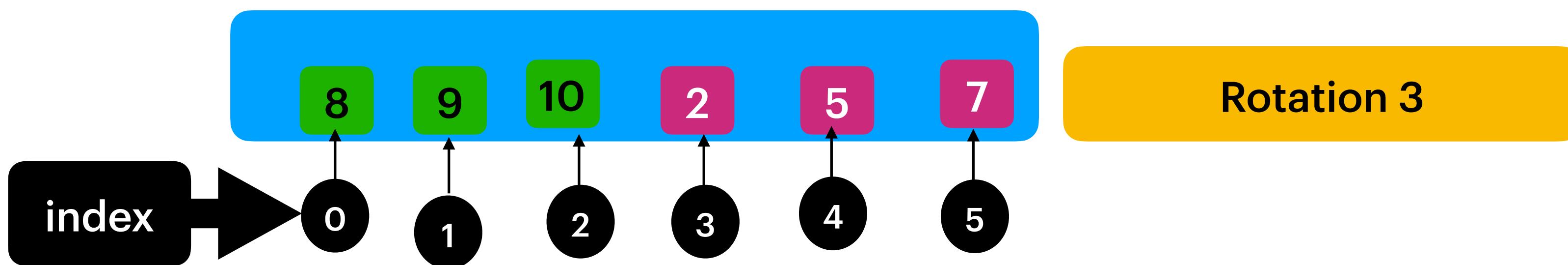
5. If arr[mid] < arr[end] , it means right array is sorted then move to left

i.e end = mid - 1;

TimeComplexity = O(logn)

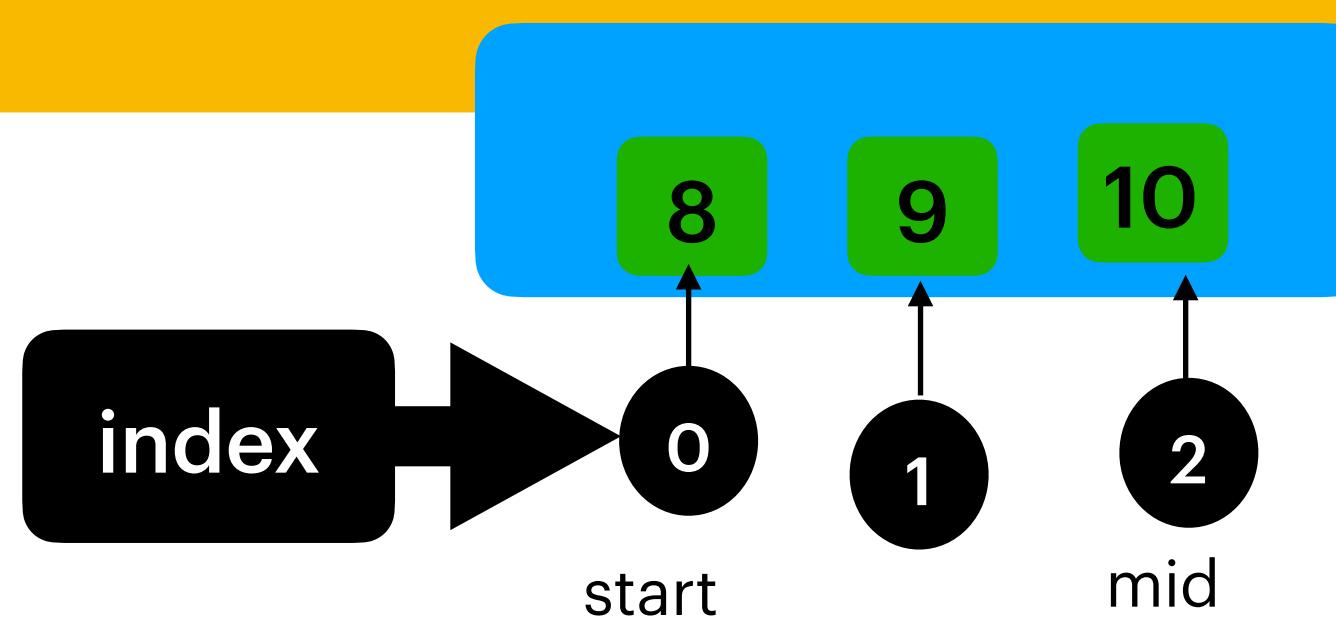
SpaceComplexity (Iteration) = O(1)

SpaceComplexity (Recursive) = O(logn)

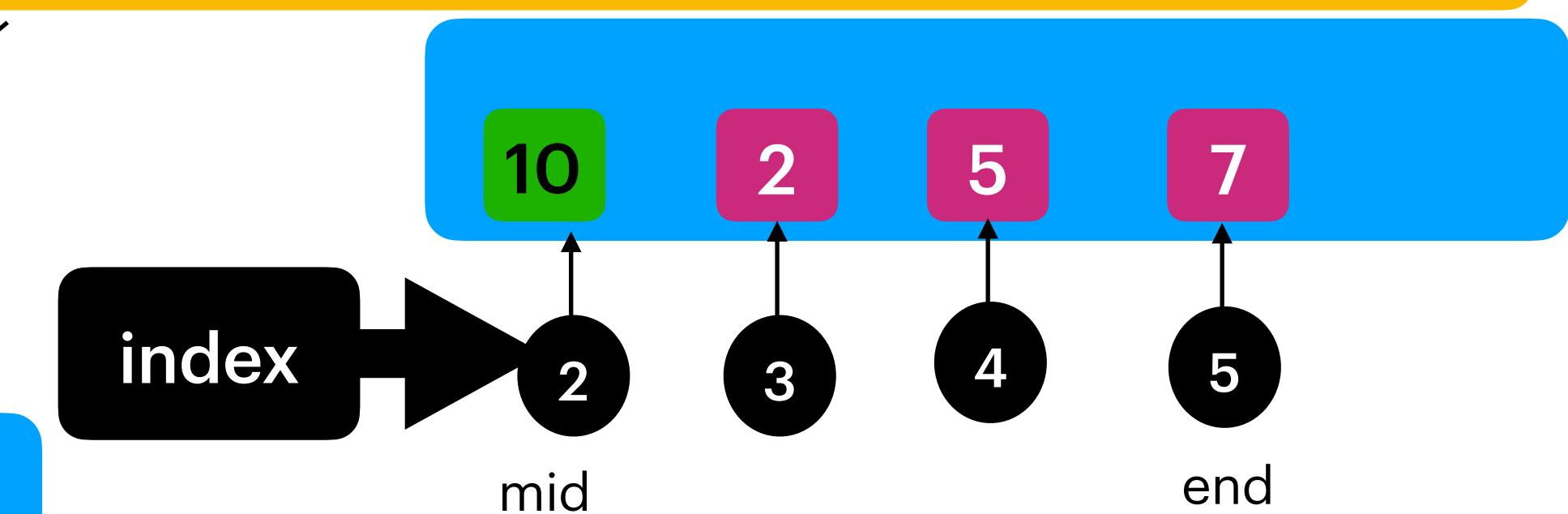


If we closely observe index "0" to mid elements are sorted.  
How ? => simple  $\text{arr}[\text{mid}] > \text{arr}[\text{start}]$

If we closely observe mid to end elements are not sorted. How ? simple  $\text{arr}[\text{mid}] \text{ is not } < \text{arr}[\text{end}]$

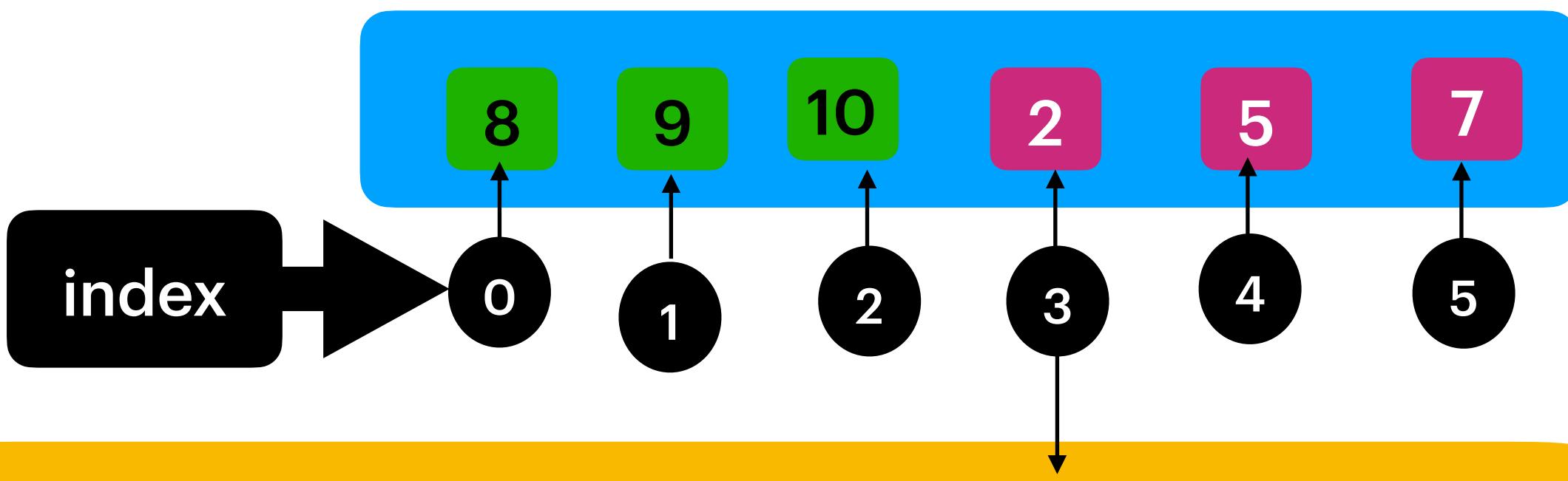


This two cases are enough to make a switch. We always find a small element in unsorted array.



If  $\text{arr}[\text{mid}] > \text{arr}[0]$ , It means left part is sorted so move to right .  
 $\text{start} = \text{mid} + 1$

If  $\text{arr}[\text{mid}] < \text{arr}[\text{end}]$  , It means right part is sorted so move to left .  
 $\text{end} = \text{mid} - 1$



### **Problem Statement :**

Find index of an element in a rotated sorted array.

Ex: {3,4,5,1,2}

targetElement = 5

output = 2 (Index of the target 5)

Ex: {3,4,5,1,2}

target = 8

output = -1

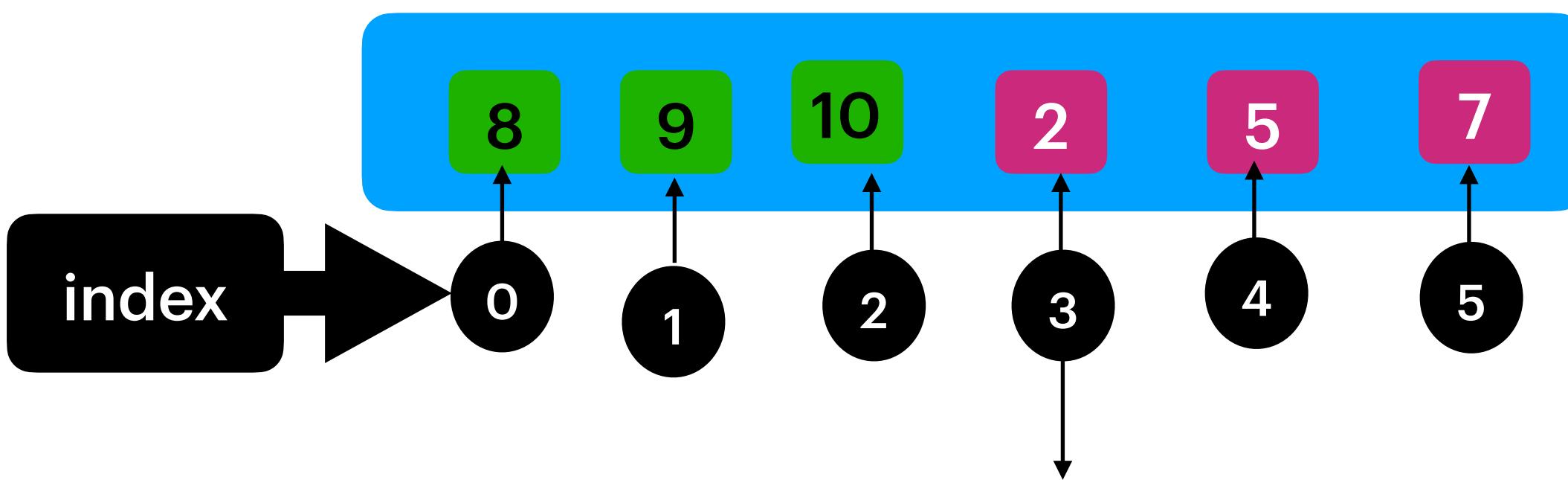
(No element found with the target 8 so return -1)

### **BruteForce Algorithm :**

LinearSearch, start from 0 repeat till n-1 , if the element found return index.

Time Complexity: O(n)

Space Complexity: O(1)



### Find a rotationIndex

```
rotationIndex == 0
search(arr, 0, n-1, target)
```

```
rotationIndex != 0
```

```
if(target >= arr[0])
    start = 0
end = rotationIndex-1
search(arr,start , end , target)
```

```
if(target <= arr[0])
start = rotationIndex
end = arr.length-1
search(arr,start , end , target)
```

### Problem Statement :

Find an element in a rotated sorted array.

Ex: {3,4,5,1,2}

targetElement = 5

output = 2 (Index of the target 5)

Ex: {3,4,5,1,2}

target = 8

output = -1

(No element found with the target 8 so return -1)

### BinarySearch Algorithm

1. Find the rotationIndex.

2. If the rotationIndex is '0' then search from index 0 to n-1

3. If the rotationIndex != 0 then we need to choose either leftPart or RightPart

If the target >= arr[0], we need to search in left part.  
i.e search(arr, 0, rotationIndex-1 , target)

If the target <= arr[0], we need to search in right part.  
i.e search(arr, 0, rotationIndex , n-1, target)

TimeComplexity = O(logn)

SpaceComplexity (Iteration) = O(1)

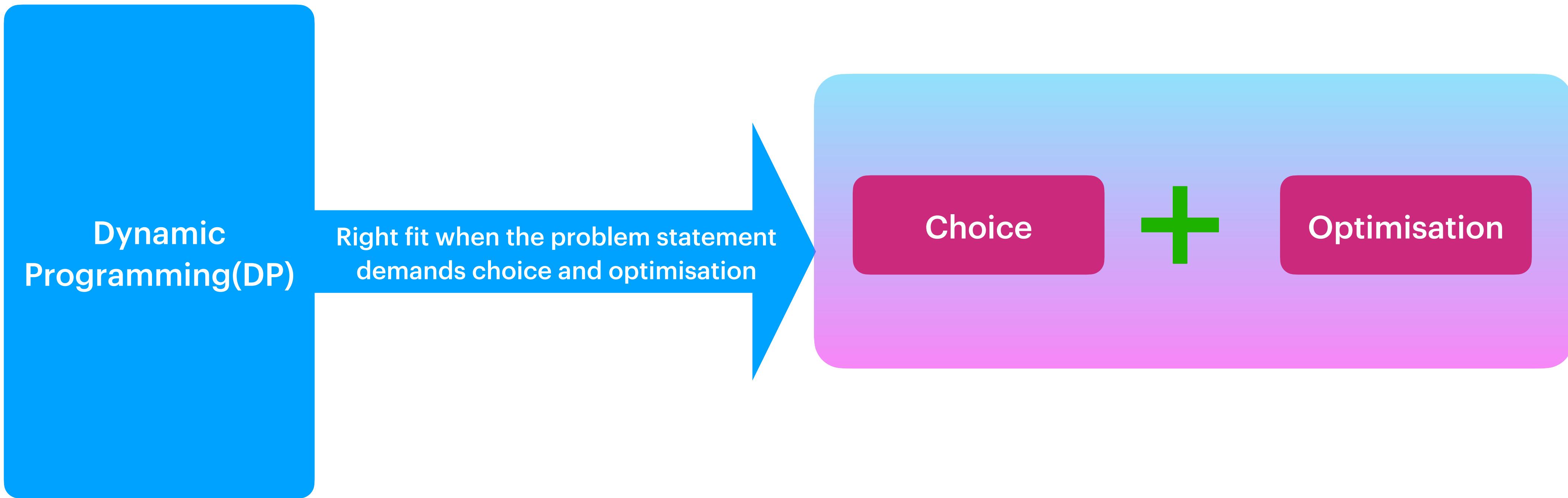
SpaceComplexity (Recursive) = O(logn)

	<b>What?</b>	<b>Why?</b>	<b>When?</b>	<b>How ?</b>
Sorting	Ordering elements in either Ascending or Descending	If we don't sort, search takes linear time $O(n)$ , with sorting we can do BinarySearch which can be done in $O(\log n)$ time.	When we sort, search would be improved.	InsertionSort MergeSort QuickSort
Binary Search	Elements less than the root on left side, greater than root on right side. {1,2, <b>3</b> ,4,5}	With BinarySearch at every step we can eliminate $n/2$ elements recursively .Helps us to find element in $O(\log n)$	We can use BinarySearch when the elements are sorted.	Find a mid element do the math over the mid for solving the problem.

**Dynamic Programming** says derive a optimised solution at “sub problem” level which obviously result to a optimistic solution for the main problem.

The best way to understand Dynamic Programming is through recursion. With recursion we can reach to the depth of the problem(i.e sub problem) from there we can give the optimised solution.

Dynamic Program is right fit when the problem statement demands Choice & Optimisation.





## Let's figure out the use cases of Dynamic Programming (DP).

### Withdraw use case in ATM:

ATM make sure always minimal number of notes would be given while withdraw process.

Let's see how this fit into DP.

ATM has three choices (100, 500, 2000 notes) and then make sure minimal notes would be given to user

Ex : Withdraw of 2000 = 1 note (i.e 2k note)

Withdraw of 1700 = 5 notes ( 500 notes 3 + 100 notes 2)

. It's a DP problem.

DP = Choice (Out of 3 Notes) + Optimisation (Should be minimal)

### Finding shortest Path in Google Maps.

In Google Maps when we choose destination, there could be multiple possible routes from source to destination but the shortest path to be chosen by Google Maps.

It's a DP problem.

DP = Choice (Out of multiple routes/nodes)  
+

Optimisation (shortest path to be chosen )

### Finding higher profit with fixed budget on stock exchange.

We have wide number of stocks available in the market but we need choose specified stocks which gives higher profit for a given budget.

It's a DP problem.

DP = Choice + Optimisation

### Travelling Salesman (Amazon Delivery Agent)

When Amazon Delivery Agent delivering products to specific locations, he should not revisit the same location twice.

It's a DP problem.

DP = Choice (locations) + Optimisation (should not be revisited)

DP says don't solve the sub problem more than once. Store the result into cache then reuse.

$f(n)$

It's a simple recursion where  $f(n)$  solving the problem by dividing into  $(n-1)$  &  $(n-2)$  sub problems. Here there is a high possibility of Overlapping. We end up with solving same problem more than once. This can be avoided with DP.

$f(n-1)$

Overlapping

$f(n-2)$

$f(n-2)$

$f(n-3)$

$f(n-4)$

$f(n-3)$

$f(n-4)$

$f(n-4)$

$f(n-5)$

$f(n-3)$

$f(n-4)$

$f(n-5)$

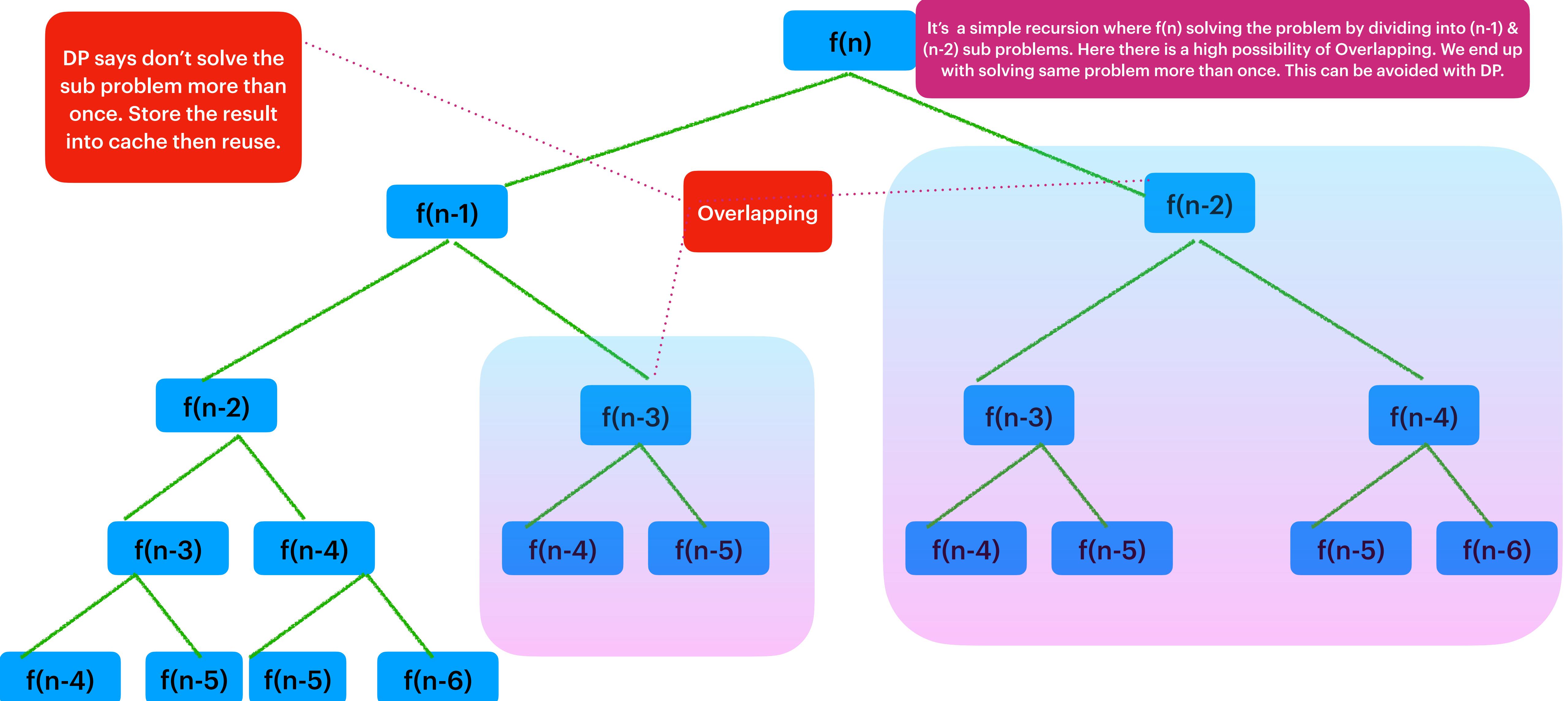
$f(n-6)$

$f(n-4)$

$f(n-5)$

$f(n-5)$

$f(n-6)$



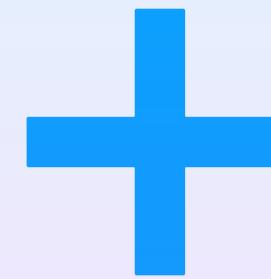
# Dynamic Programming

Two ways to solve the DP problems

Memoization (Top-down)

Tabulation (Bottom -Up)

Recursion



Storage

Iterative Approach



Storage

## Dynamic Programming

### Recursion

Step1

Recursion



Cache Sub  
Problem solutions  
then reuse.

Memoization

Step2

Iterative Approach



Cache Sub Problem  
solutions then reuse.

Tabulation

Step3

We have Items, every Item has weight and profit.

## Knapsack (Bag) Problems

Fixed Capacity

**Problem Statement :** Every Knapsack has the capacity, we should choose the items , which can give higher profit, by considering capacity in mind.

example : itemWeight = {1,2,3} profitProfit = {5,2,4} capacity : 5

### Dynamic Programming Solutions

Allow only unique items.

#### 0/1 Knapsack

we either take the whole item or don't take it.

#### Unbound Knapsack

Allow duplicate items.

#### Greedy Algorithm

Allow only unique items.  
But we can split, the item.

#### Fractional Knapsack

Item weight {1,2,3}  
Item Profits {5,2,4}  
Capacity : 5  
Max Profit we can gain by choosing items {1,3} & respective profits {5,4} = 9

Capacity left in bag is 1. As we chosen Item Weights {1,3} = 4



Capacity left in bag is 0. As we chosen item 1, five times.

Item weight {1,2,3}  
Item Profits {5,2,4}  
Capacity : 5  
Max Profit we can gain by choosing item {1} 5 times & profit is  $5 * 5 = 25$ .



Capacity left in bag is 0.

Item Weights {1,2,3}  
Item Profits {5,2,4}  
Capacity : 5  
item1 + 50% of weight of item2 + item3 = then profit is  $= 5 + 1 + 4 = 10$

We have Items, every Item has weight and profit.

## 0/1 Knapsack Problems

Fixed Capacity

**Problem Statement :** Given item weights and their respective profits, itemWeight = {1,2,3} ItemsProfit = {5,2,4} , Get the max profit.

**Constraints :**

The knapsack capacity : 5

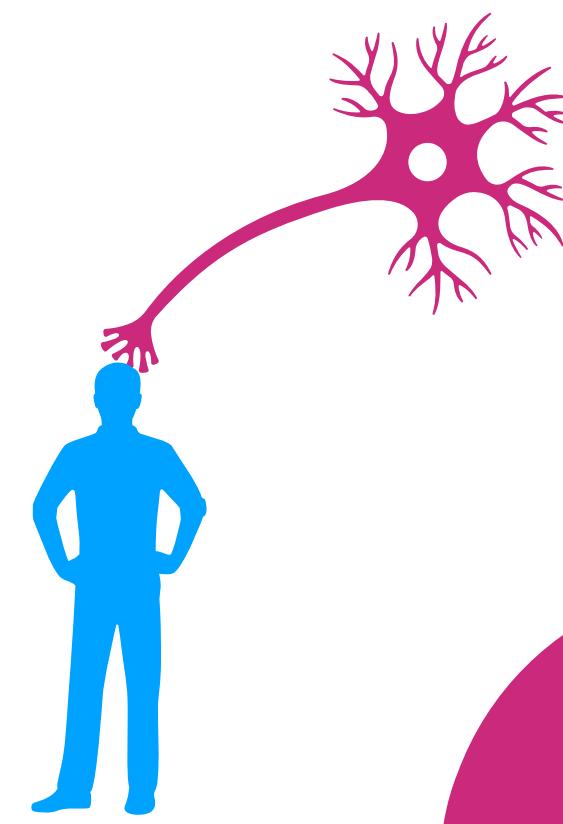
We allowed to choose the item only once.

**Expected Output :**

Max Profit we can gain by choosing itemWeights {1,3} & respective profits {5,4} = 9

Max Profit = 9

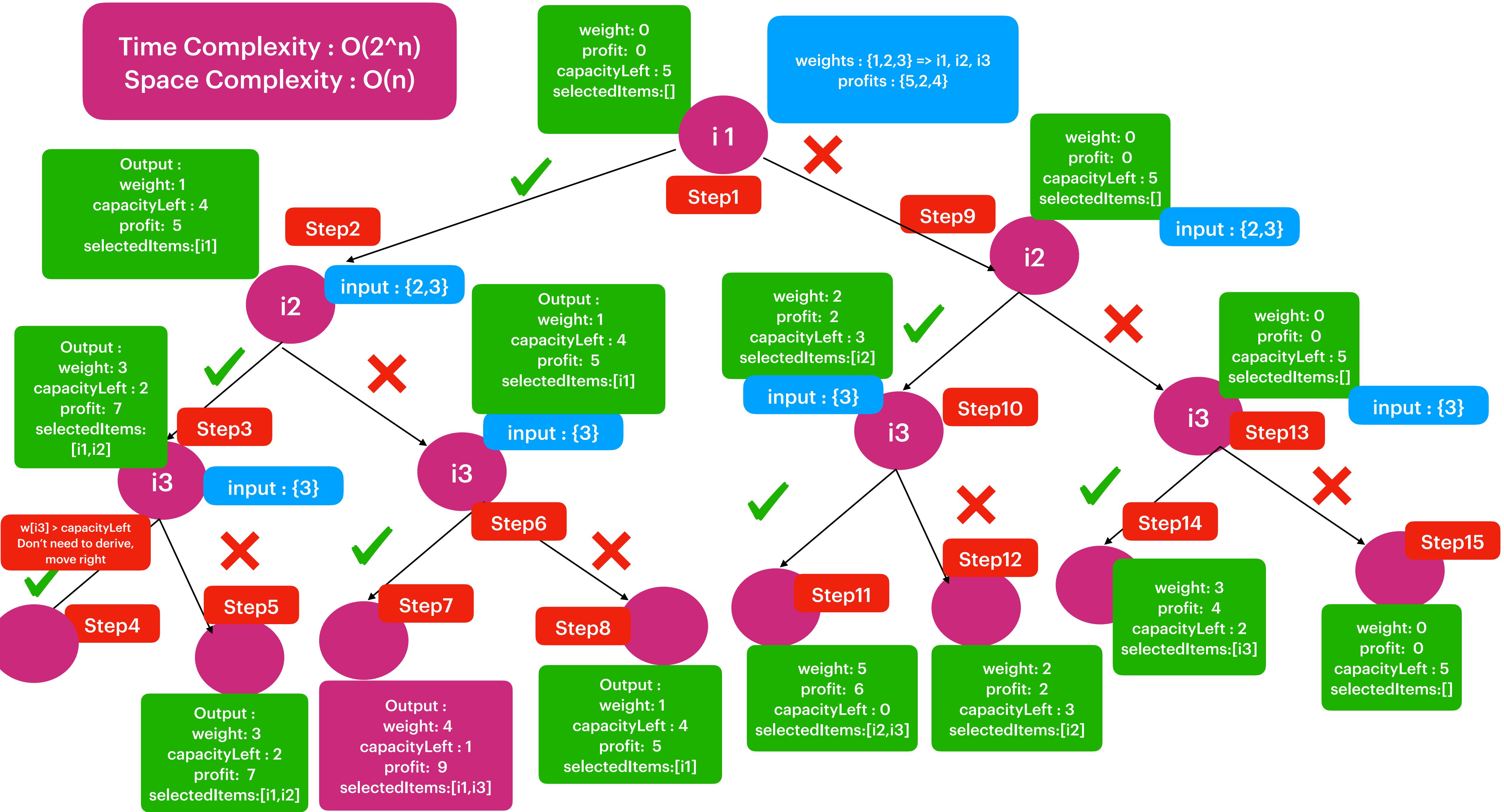
Selected items Weight  
{1,3}



Let's figure out all possible permutations.  
itemsWeight = {1,2,3}  
Capacity = 5  
ItemsProfit = {5,2,4}

- 1) Don't take any item => profit = 0, weight=0, capacityLeft = 5
- 2) i1 w{1} , profit {5} ,capacityLeft = 5-1 = 4
- 3) i2 w{2}, profit {2}, capacityLeft = 5 - 2 = 3
- 4) i3 w{3}, profit {4}, capacityLeft = 5 - 3 = 2
- 5) i1, i2 w{1+2 = 3}, profit{5+2 = 7} ,capacityLeft = 5 - 3 = 2
- 6) i1, i3 w{1+3 = 4}, profit{5+4 = 9} ,capacityLeft = 5 - 4 = 1
- 7) i2, i3 w{2+3 = 5}, profit{2+4 = 6}, capacityLeft = 5 - 5 = 0
- 8) i1, i2, i3 w{1+2+3 = 6} w > capacity ; 6 > 5 don't derive logic

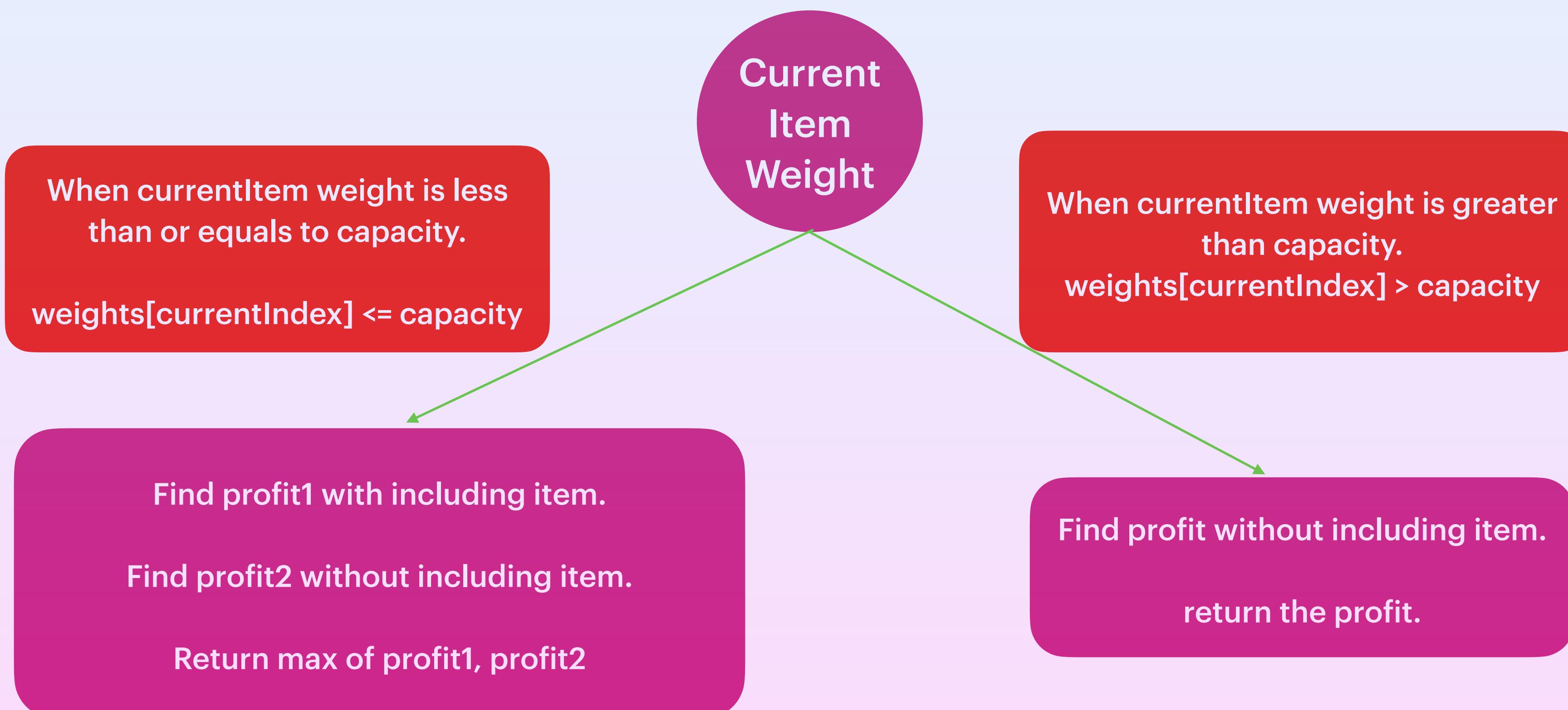
Time Complexity :  $O(2^n)$   
Space Complexity :  $O(n)$

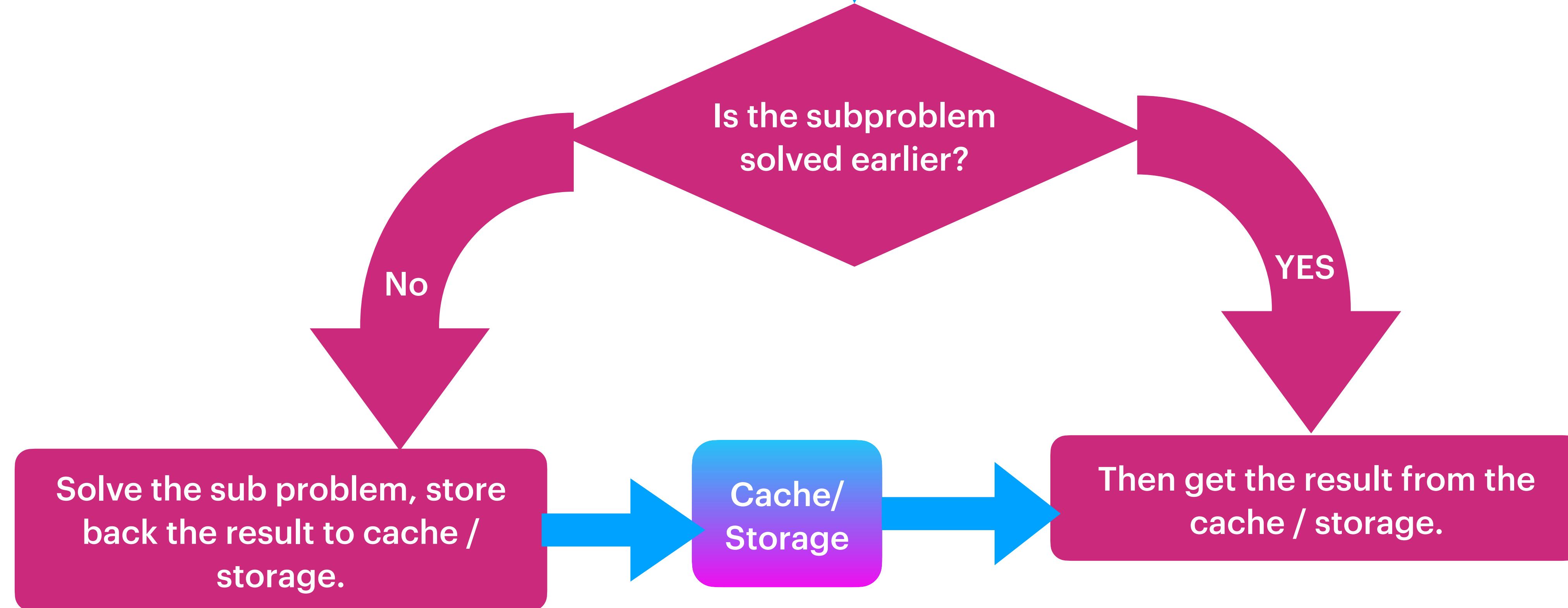
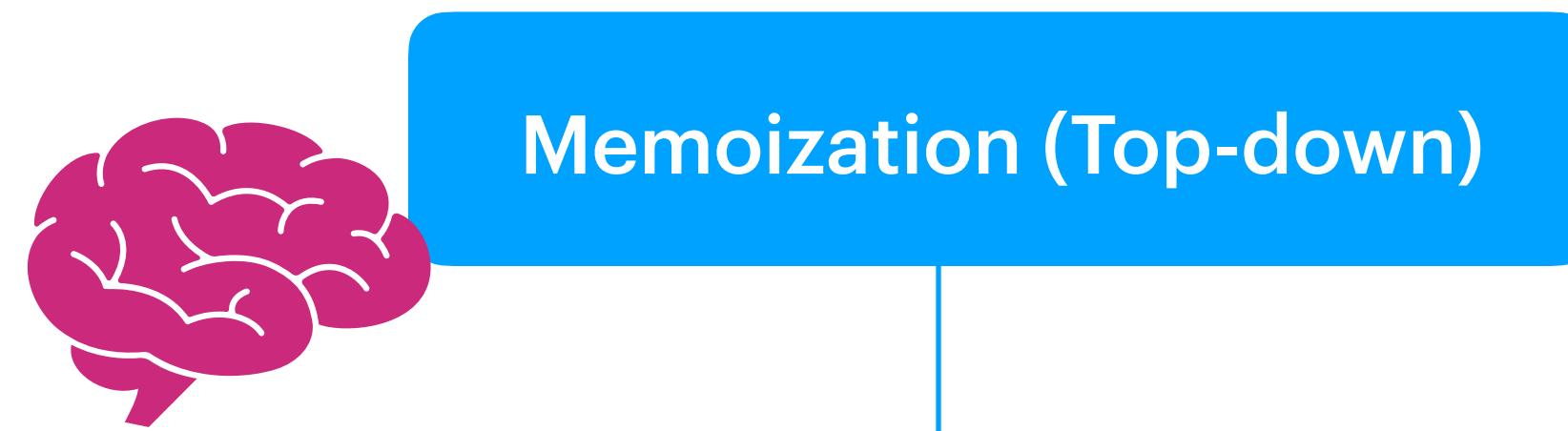


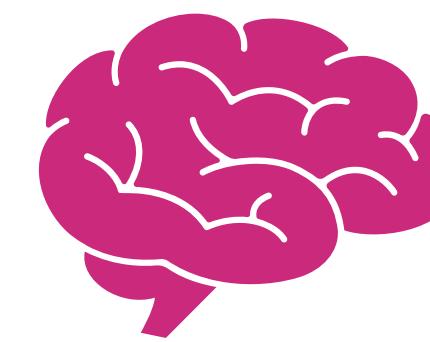
# Algorithm For Recursion

Base Condition in Recursion :

1. As we are moving from index 0 to n, the max possible valid value for `currentIndex = n-1`
2. As and when we add item to the knapsack, we reduce the capacity so the smallest possible value for capacity is '0'.







## Memoization (Top-down)

How to identify  
the sub problems for caching  
/ storing?

Have a proper knowledge on the data structure design, observe what are the parameters are being changed & encouraging us to move to all possible sub problems.

**Hint :** In simple whatever parameters we use in base condition of recursion, can be used to identify the caching / storing.

Memoization (Top-down)

## 0/1 Knapsack Problems



For the 0/1 Knapsack problems, there are two parameters, can encourage us to move forward with possible sub problems.

1. currentIndex (moving from index 0 to n-1)
2. capacity ( Moving from capacity value "c" to 0)

So finally we can have a storage/caching on these two variable for all possible combinations.

We can represent a matrix with two dimensional array.

Matrix is the place where we can design all possible combinations.

All possible combinations?  
How ?



Matrix 2 \* 3

2 4 6  
3 5 7

I can represent a Matrix as  
two dimensional array.  
{ {2,4,6}, {3,5,7} }

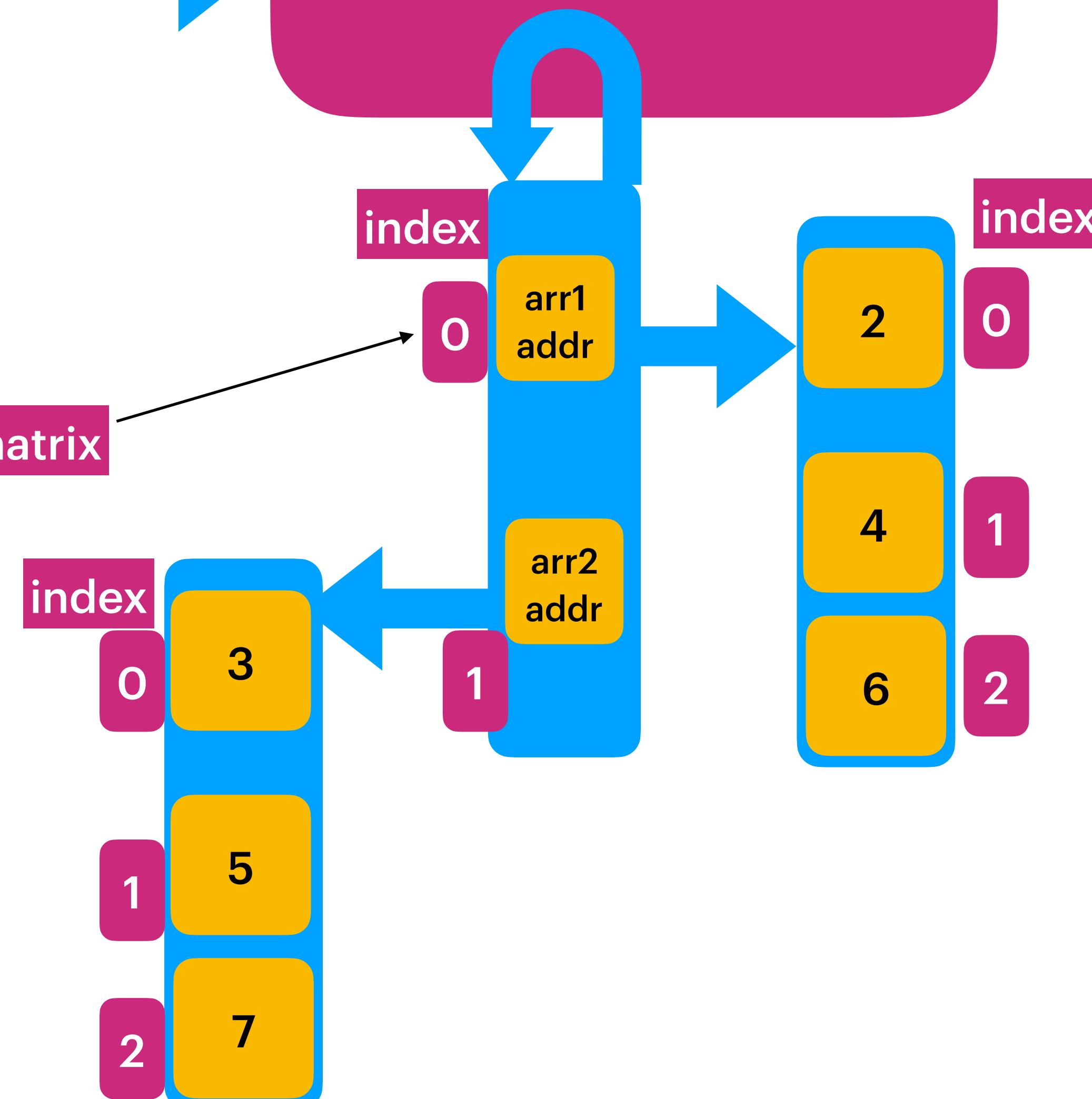
```
int[][] matrix = { {2,4,6}, {3,5,7} };
```

matrix

matrix[0] => arr1addr  
matrix[1] => arr2addr

matrix[0][0] = arr1addr[0] = 2  
matrix[0][1] = arr1addr[1] = 4

matrix[1][0] = arr2addr[0] = 3  
matrix[1][1] = arr2addr[1] = 5



# Algorithm For 0/1 Knapsack Memoization

design a matrix with  
currentIndex and  
capacity.

`int[][] dp = new int[n][capacity+1];`

Which is  
representing  
currentIndex.  
Index starts  
from 0 to n-1

Solve the sub  
problem for the  
current  
combination of  
index & capacity.  
Store back the  
result to Matrix.

- Base Condition in Recursion :
1. As we are moving from index 0 to n, the max possible valid value for currentIndex = n-1
  2. As and when we add item to the knapsack, we reduce the capacity so the smallest possible value for capacity is '0'.

Check in matrix, does this  
combination of (index, capacity) sub  
problem is already been solved?

YES

Just return the result from  
the matrix.

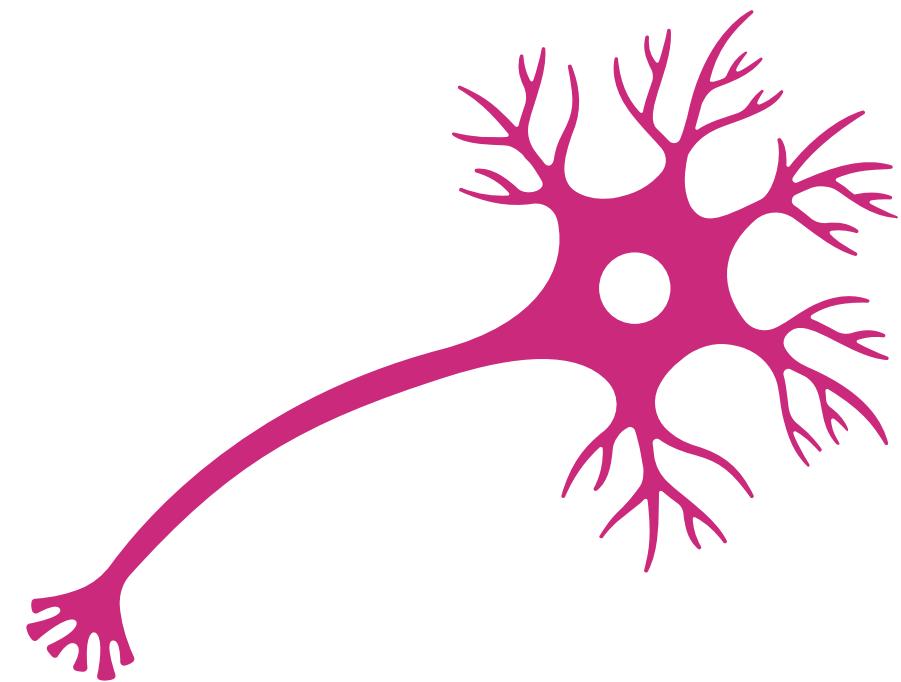
When currentItem weight is less  
than or equals to capacity.  
 $\text{weights[currentIndex]} \leq \text{capacity}$

Find profit1 with including item.  
Find profit2 without including item.  
Return max of profit1, profit2

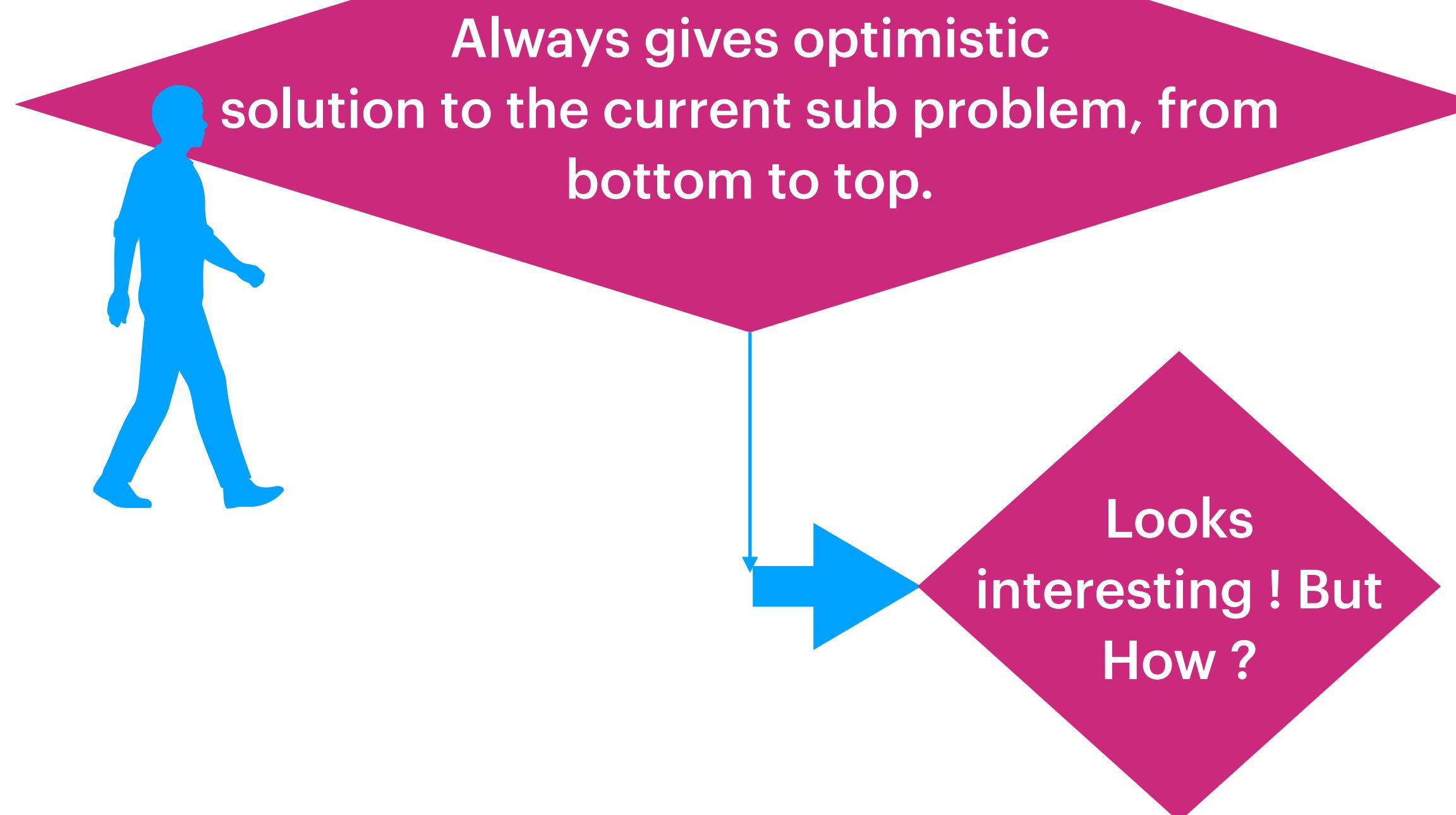
Current  
Item  
Weight

When currentItem weight is greater  
than capacity.  
 $\text{weights[currentIndex]} > \text{capacity}$

Find profit without including item.  
return the profit.



## Tabulation (Bottom-UP Approach)

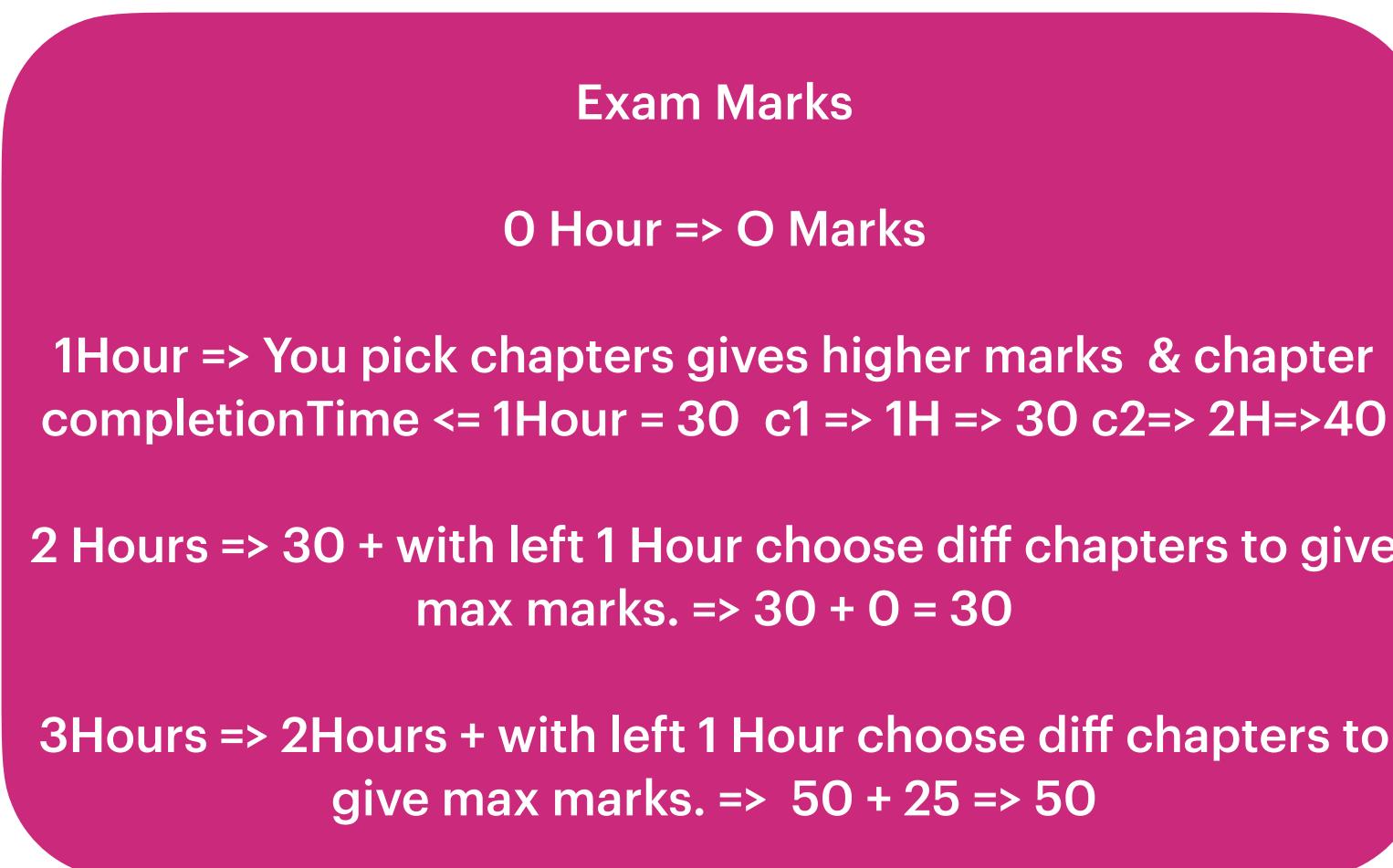


## Analogy For Tabulation.



Preparing for a exam

Goal is to make best use of current time get the target marks.



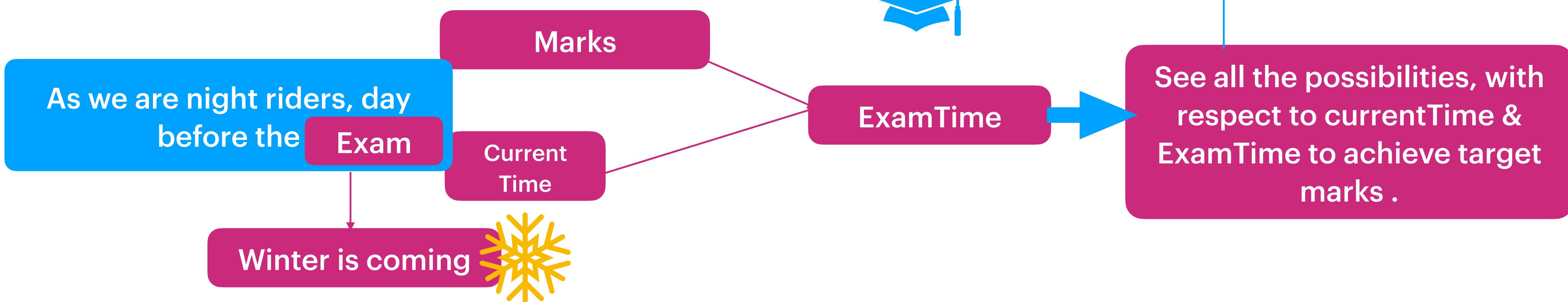
Now you divide the complete book to into two parts.

1. Time takes to complete chapter
2. Number of marks you could get from the chapter.

Here in this analogy

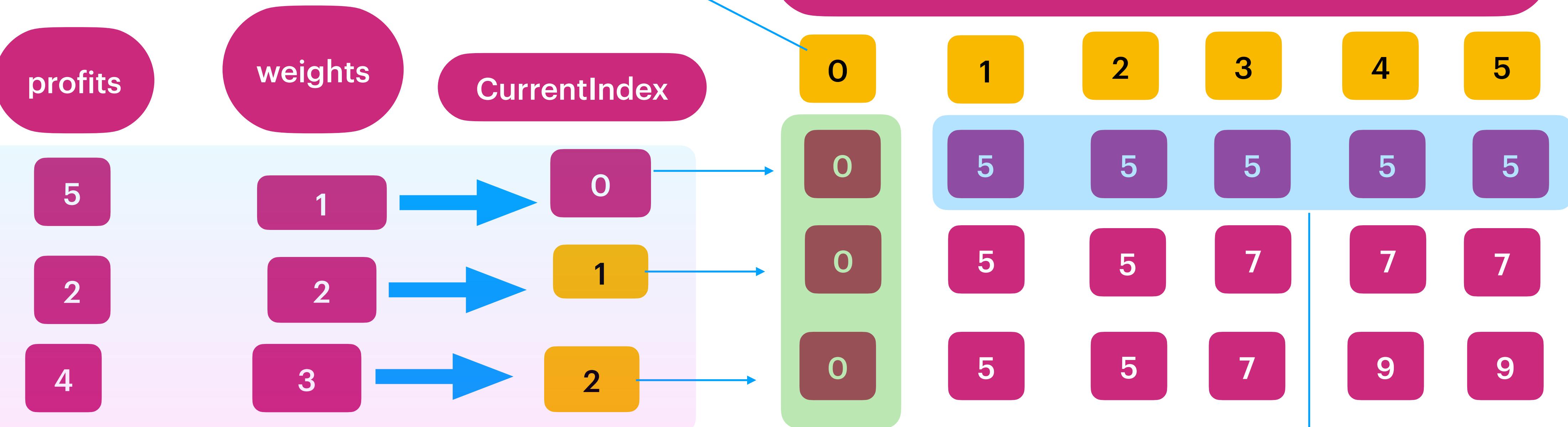
1. currentTime => Weight
2. chapterMarks => profit
3. chapterWeight => timeTakes to completeChapter
4. ExamTime => capacity

Make best use of time to get maxProfit.



For  $c=0$  capacity all the items gives 0 profit.

Capacity => 0 to 5



When currentItem weight is less than or equals to capacity.  
 $\text{weights[currentIndex]} \leq \text{capacity}$

From index=2 onwards for every use case there are 2 possibilities.

When currentItem weight is greater than capacity.  
 $\text{weights[currentIndex]} > \text{capacity}$

As we are at index 0 , It's going to be the bottom of the problem. We should start the solution from here.

Then How ? Think this way, you have only one item.  
 $\text{weights[0]} \leq \text{capacity}$  then include profit else profit would be zero.

Find profit1 with including item.  
 $\text{profit1} = \text{profit}[i] + \text{dp}[i-1][c - \text{weight}[i]]$

Find profit2 without including item.  
 $\text{profit2} = \text{dp}[i-1][c - \text{weight}[i]]$

Return max of profit1, profit2

Find profit without including item.  
 $\text{profit} = \text{dp}[i-1][c]$   
 return the profit.

## Knapsack Problem

weights {1,2,3}

profits {5,2,4}

capacity : 2

`weights[element] <= capacity`

Element

`weights[element] > capacity`

Element adds into the bag

Element can't be included into the bag

calculate profit with including current item

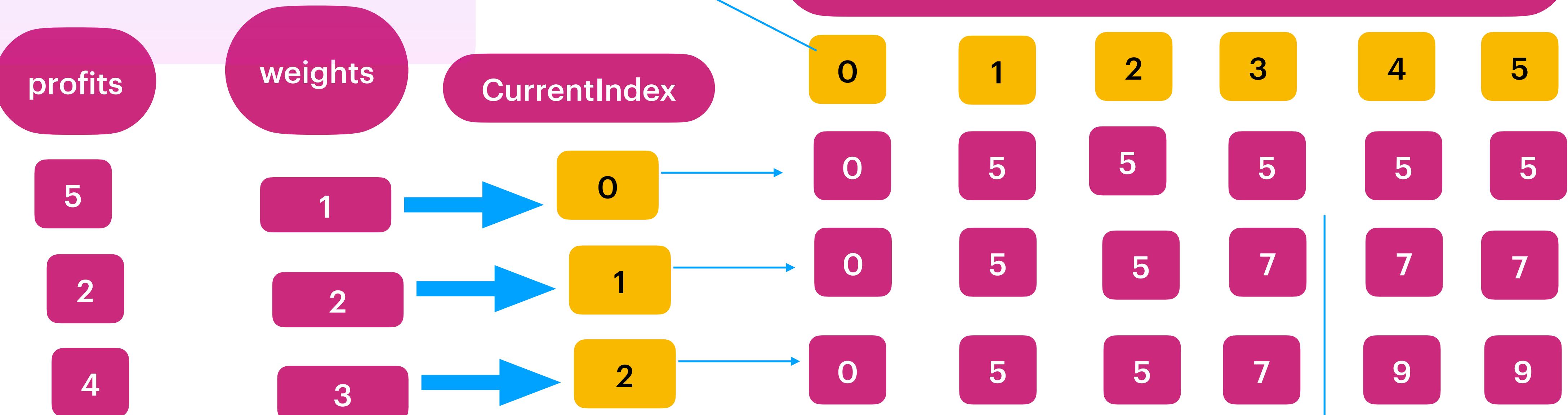
calculate profit without including current item

Just calculate profit without including current element.

Get max profit then return

For  $c=0$  capacity all the items gives 0 profit.

Capacity => 0 to 5



When currentItem weight is less than or equals to capacity.  
 $\text{weights[currentIndex]} \leq \text{capacity}$

From index=2 onwards for every use case there are 2 possibilities.

When currentItem weight is greater than capacity.  
 $\text{weights[currentIndex]} > \text{capacity}$

As we are at index 0 , It's going to be the bottom of the problem. We should start the solution from here.

Then How ? Think this way, you have only one item.  
 $\text{weights[0]} \leq \text{capacity}$  then include profit else profit would be zero.

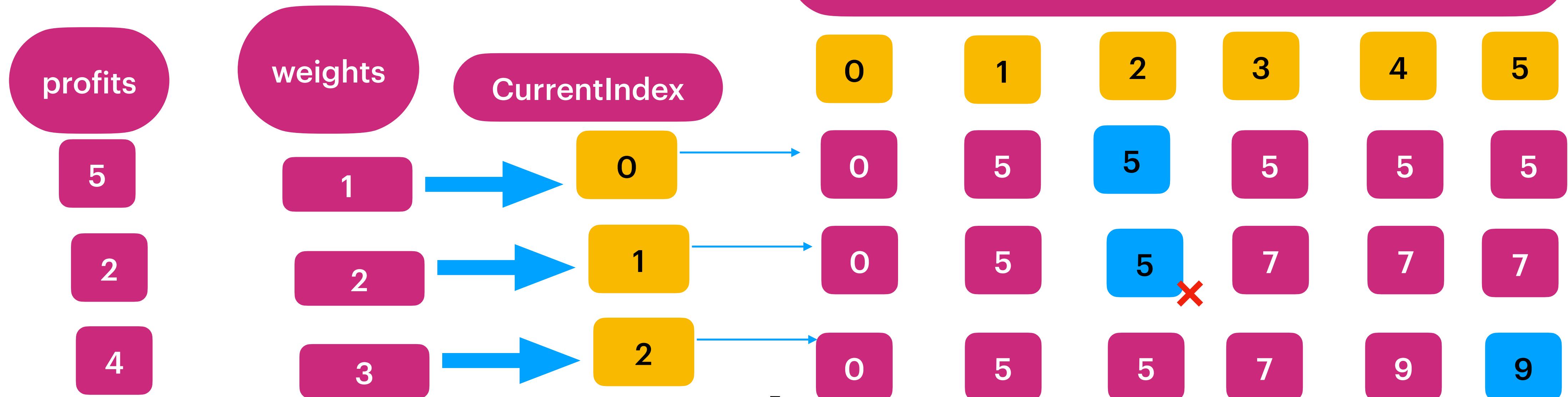
Find profit1 with including item.  
 $\text{profit1} = \text{profit}[I] + \text{dp}[i-1][c - \text{weight}[i]]$

Find profit2 without including item.  
 $\text{profit2} = \text{dp}[i-1][c - \text{weight}[i]]$

Return max of profit1, profit2

Find profit without including item.  
 $\text{profit} = \text{dp}[i-1][c]$   
return the profit.

Capacity => 0 to 5



```
int i = weights.length - 1;
int capacity = 5
int topProfit = 9
dp[][] ...

while(i > 0){
if(totalProfit != dp[i - 1][c])
{
take => weight[i] => 3
totalProfit = 9 - 4 = 5
capacity => 5 - 3 = 2
}
i --; l=0
}
```

dp[0][c] == totalProfit => dp[0][2] == 5 => weight[0] => 1

9 != 7

Take itemWeight[3]  
reduce the profit =  $9 - 4 = 5$   
reduce capacity =  $5 - 3 = 2$

## Knapsack Problem

weights {1,2,3}

profits {5,2,4}

capacity : 2

`weights[element] <= capacity`

Element

`weights[element] > capacity`

Element adds into the bag

Element can't be included into the bag

calculate profit with including current item

calculate profit without including current item

Just calculate profit without including current element.

Get max profit then return



Problem Statement : Given a set of positive numbers, determine if a subset exists whose sum is equals to a given number 'S'.

Ex : {1,3,5} , S = 8

Expected Output :

Ex: True. Its possible with {3,5}

Ex: {1,3,5}, S = 8

Let me take up the all the possible combination subsets for the given input {1,3,5}

Total Combinations 8.  
As array size = 3  
we got  $2^3$  combinations.

With empty Set always Zero

{ } => sum = 0

Choosing 1 element at a time

{1} => sum = 1

{3} => sum = 3

{5} => sum = 5

Choosing 2 elements at a time

{1,3} => sum = 4

{1,5} => sum = 6

{3,5} => sum = 8

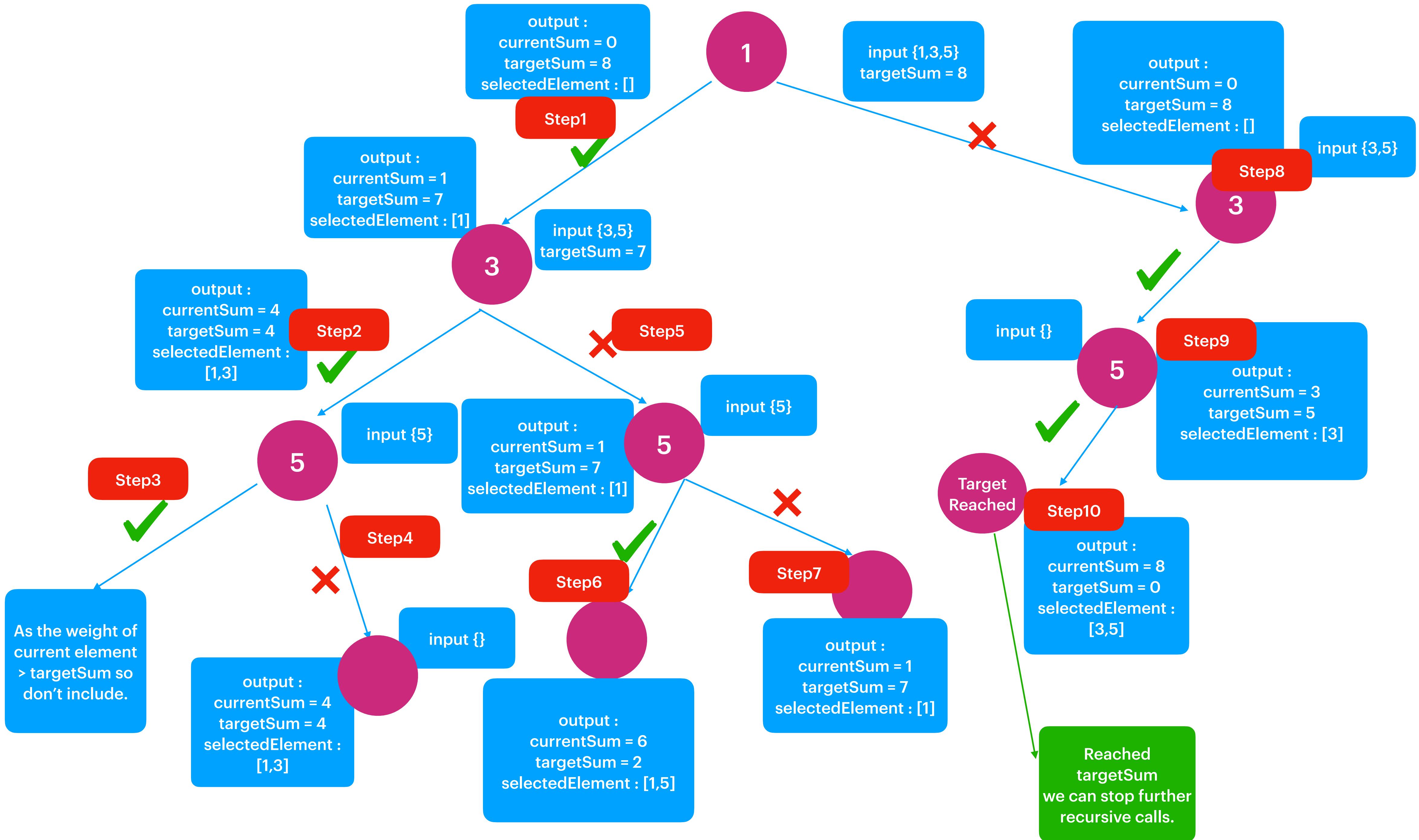
Choosing 3 elements at a time

{1,3,5} => sum = 9

As the targetSum is 8, we can with subset {3,5}

How to solve ? To get into the solution, we should see all the possible combinations (i.e) subproblems.

Solution1 => Recursion



```
int[] arr = {1,3,5}  
int targetSum = 8
```

From the given input we can derive possible sub problems are  $2^4 \Rightarrow (3 * 8)$ . We can represent in matrix.

```
Boolean[][] dp = new Boolean[3][8+1]
```

Should be 8+1, as index ends with n-1

When targetSum=0 , for any array size its true, because we can return empty subset.

Tabulation Follows bottom - up approach.

Interesting !!! So that we will derive a solution for every element in an array, from target(0) to target(n), here n = 8

1

Let's derive a optimistic solution to 1st element, Here we have only one option left. If the sum == arr[0] then return True else False

Target Sum										
Value	Index	0	1	2	3	4	5	6	7	8
1	0	T	T	F	F	F	F	F	F	

Now we got all possible solutions from target(0) to target(n), these subproblem solutions we can use in next steps.

```
int[] arr = {1,3,5}  
int targetSum = 8
```

3

Let's derive optimistic solution to 2nd element, Here we have two options.

Current Element

arr[index] <= capacity



arr[index] > capacity

Include the element

Exclude the element

calculate the possibility  
with including current  
element

calculate the possibility  
without including current  
element

dp[i-1] [ targetSum - arr[i] ]

||

dp[i-1] [ targetSum ]

Just calculate possibility without  
including current element.

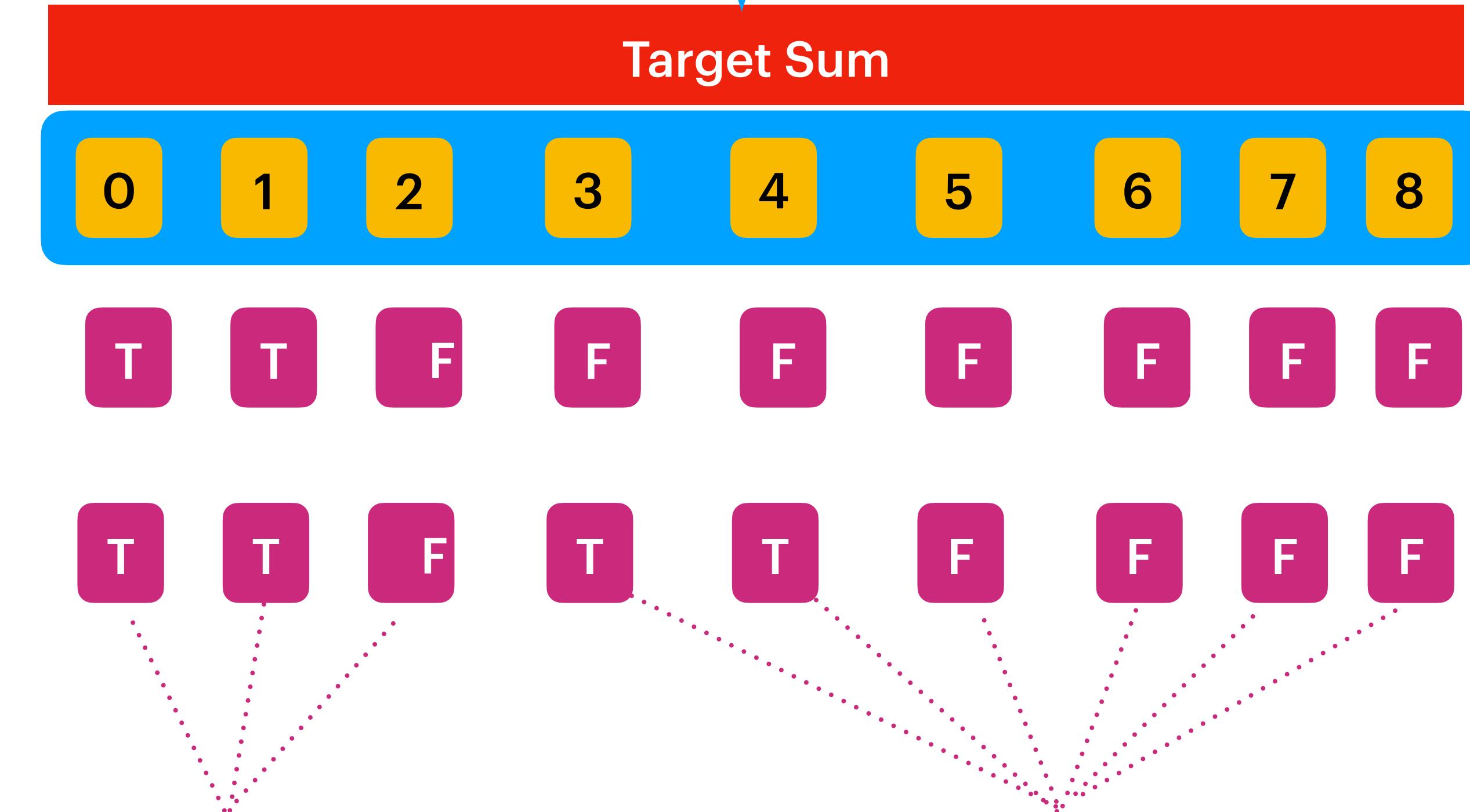
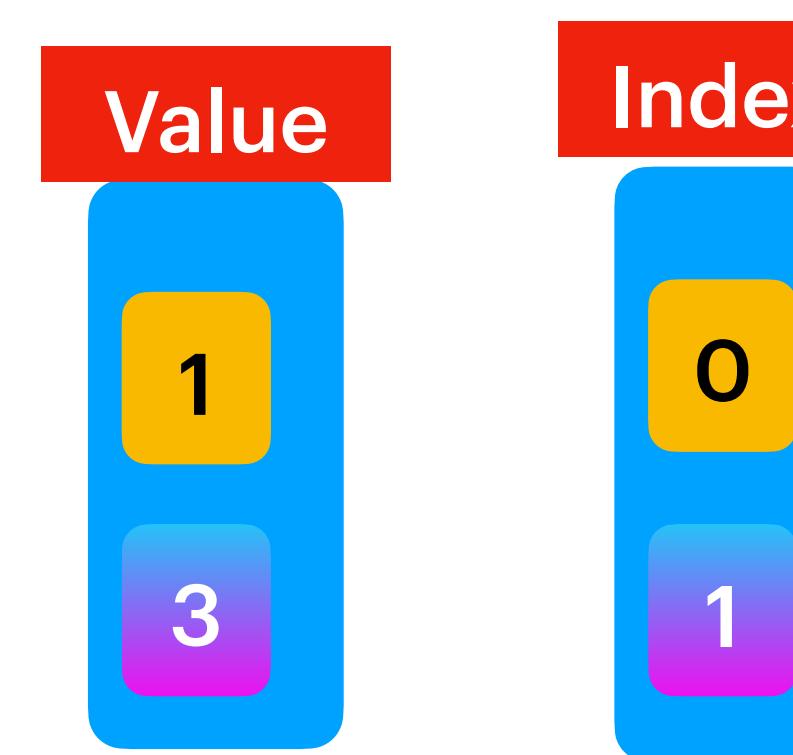
return dp[i-1] [ targetSum ]

return true, either of the one is true . (Use || )

```
int[] arr = {1,3,5}  
int targetSum = 8
```

3

Now, It's time to derive optimistic solution to 2nd element.



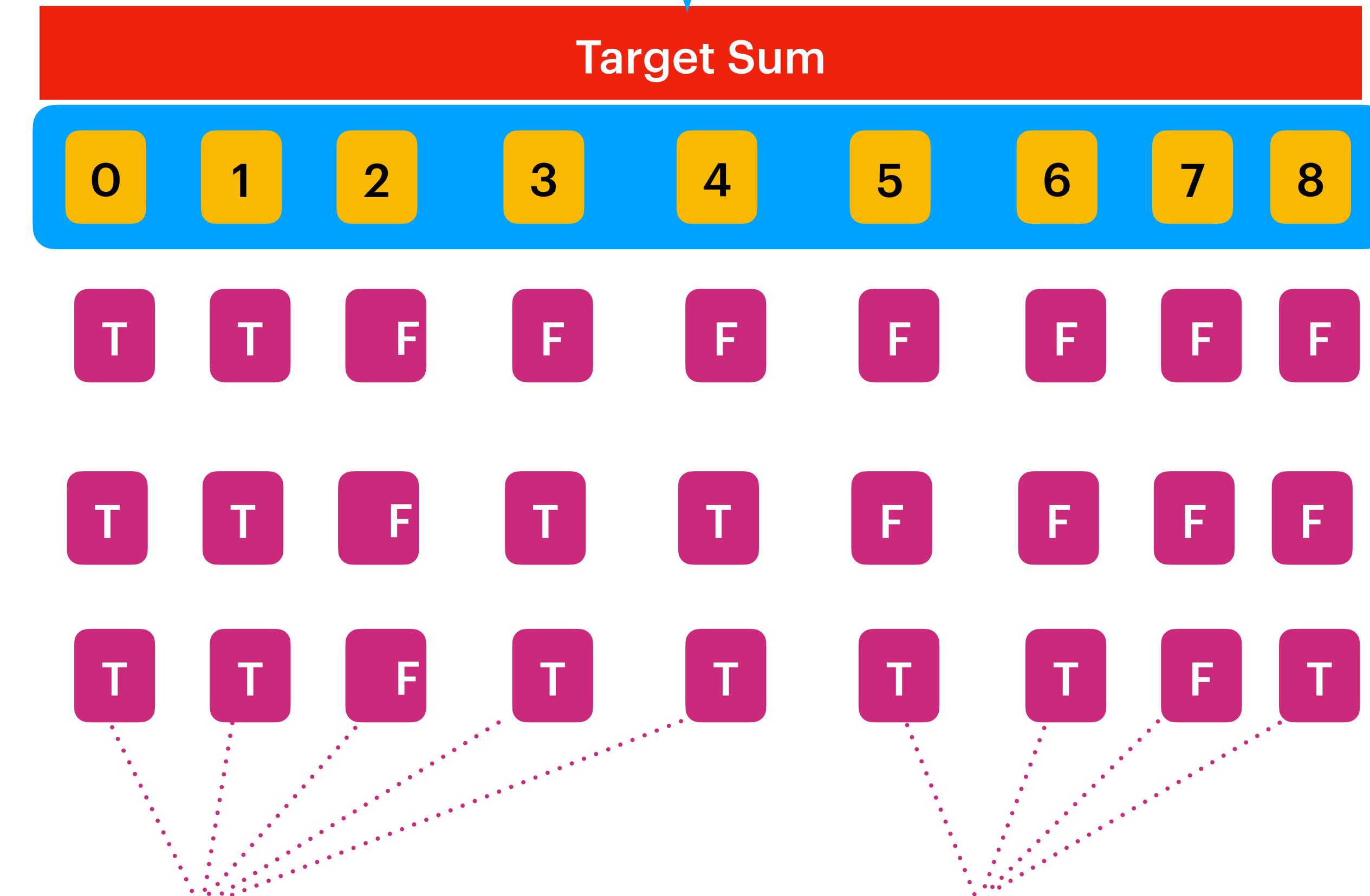
Till here the current element (3) > respective target sums (0,1,2) so we taken  
respective values from previous sub problem. (i.e from 0 index).  
return dp[0][sum]

From sum(3) onwards the current element (3) <= respective  
targetSum so  
return dp[0][sum-arr[1]] || dp[0][sum]

```
int[] arr = {1,3,5}  
int targetSum = 8
```

5 Now, let's figure optimistic solution to 3rd element. Procedure is same as finding optimistic solution for 2nd element.

Value	Index
1	0
3	1
5	2



Till sum(4) the current element (5) > respective target sums (0,1,2,3,4) so we taken respective values from previous sub problem. (i.e from index 1).

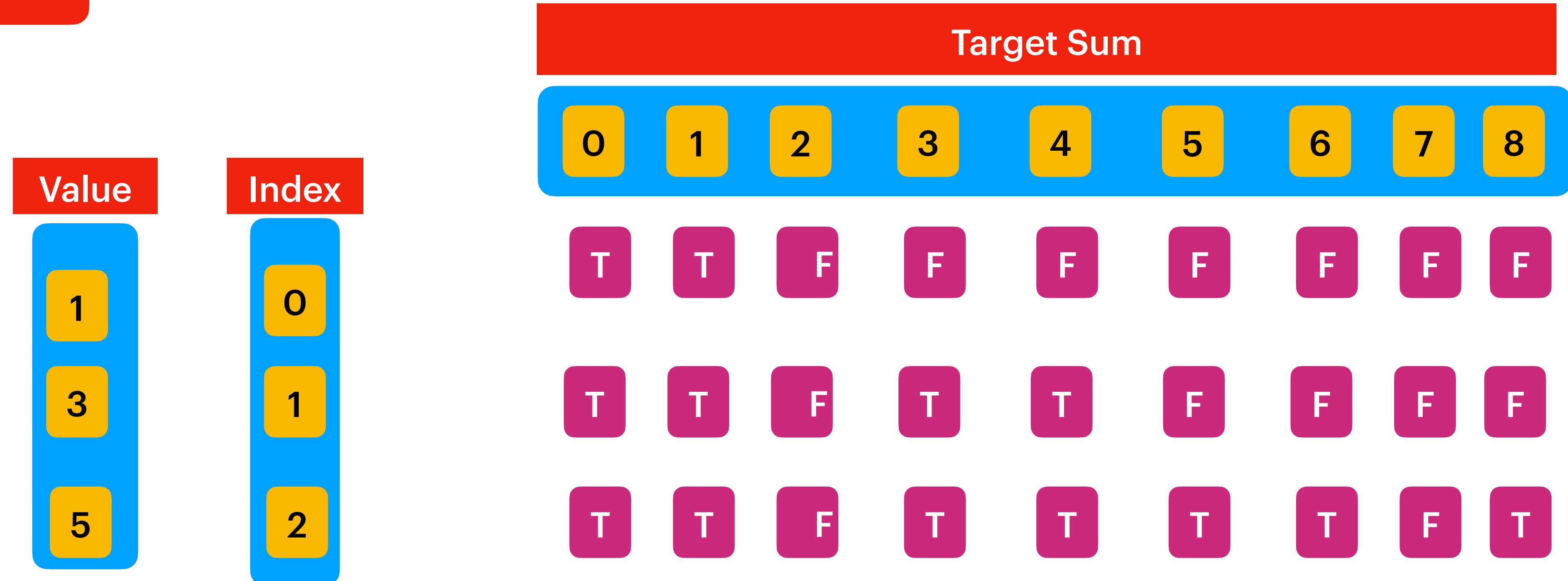
```
return dp[1][sum-arr[1]] || dp[1][sum]
```

From sum(5) onwards the current element (5)  $\leq$  sum so we

```
return dp[1][sum-arr[1]] || dp[1][sum]
```

```
int[] arr = {1,3,5}  
int targetSum = 8
```

Finally we rocked it, derived a solution for targetSum 8 with given input elements {1,3,5}.



Problem Statement : Given a set of positive numbers, assume that always subset exist with given sum 'S', so print the possible subset.

Ex : {1,3,5} , S = 8

Expected Output : {3,5}

Ex: {2,3,5}. , S = 5

output : {2,3}, {5}

Back Tracking .....

		Target Sum									
Value	Index	0	1	2	3	4	5	6	7	8	9
1	0	T	T	F	F	F	F	F	F	F	F
3	1	T	T	F	T	T	F	F	F	F	F
5	2	T	T	F	T	T	T	T	F	T	T

#### Algorithm:

It's a Back Tracking problem. Move to the top of the record. In our example i=2, sum = 8 . dp[2][8] = True

1. In our use case when  $dp[i][sum]$  is true, we would need to check does it copied from previous problem or not.  
we have two possibilities here

- 1) when  $dp[i-1][sum]$  is false and  $dp[i][sum]$  is true, it means we includes the current "i".
- 2) when  $dp[i-1][sum]$  is true and  $dp[i-1][sum - arr[i]]$  is true, it means we includes the current "i"

## **Equal Subset Sum Partition :**

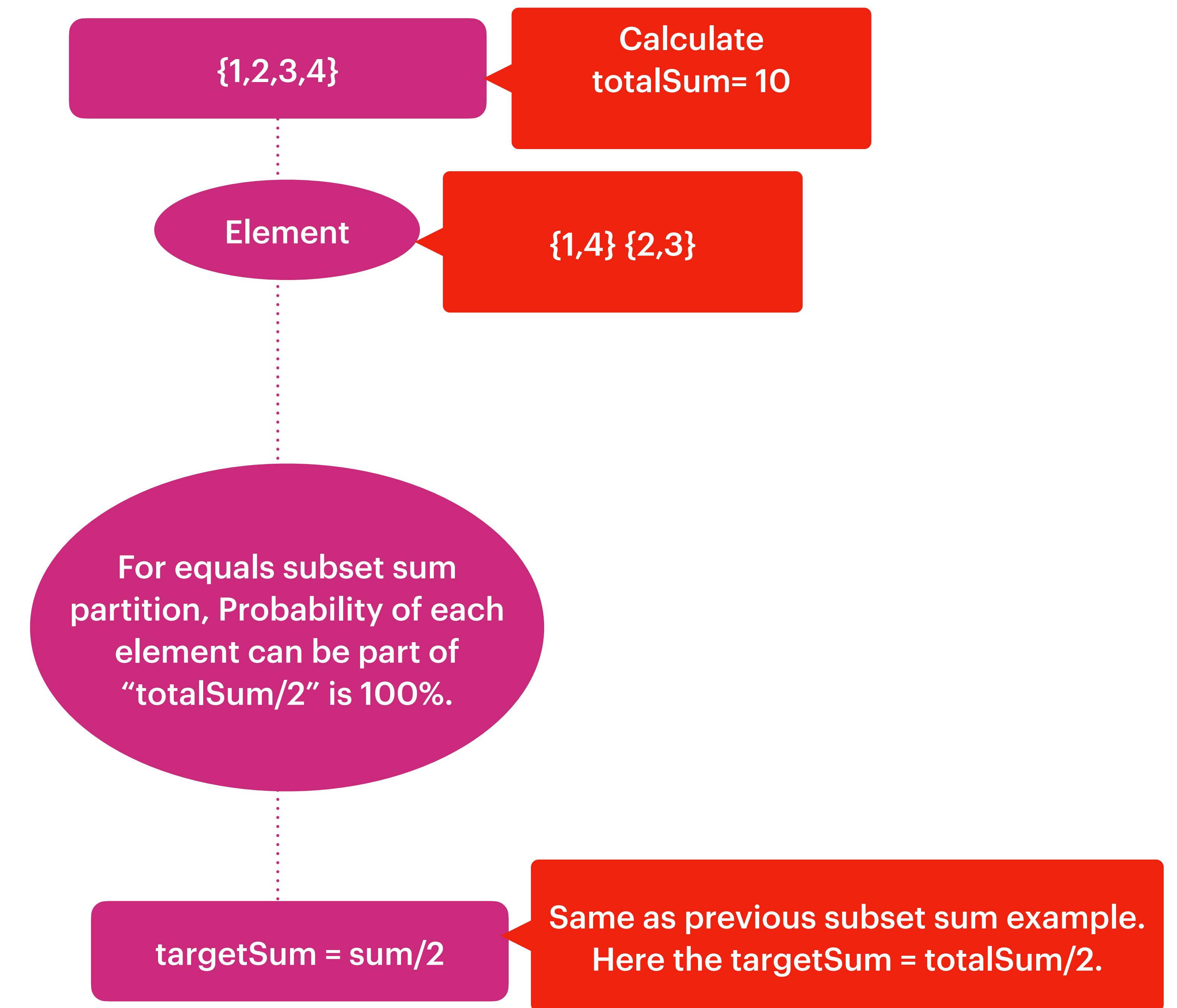
**Given a set of positive numbers, find if we can partition it into two subsets such that the sum of elements in both the subsets is equal.**

**Ex:**

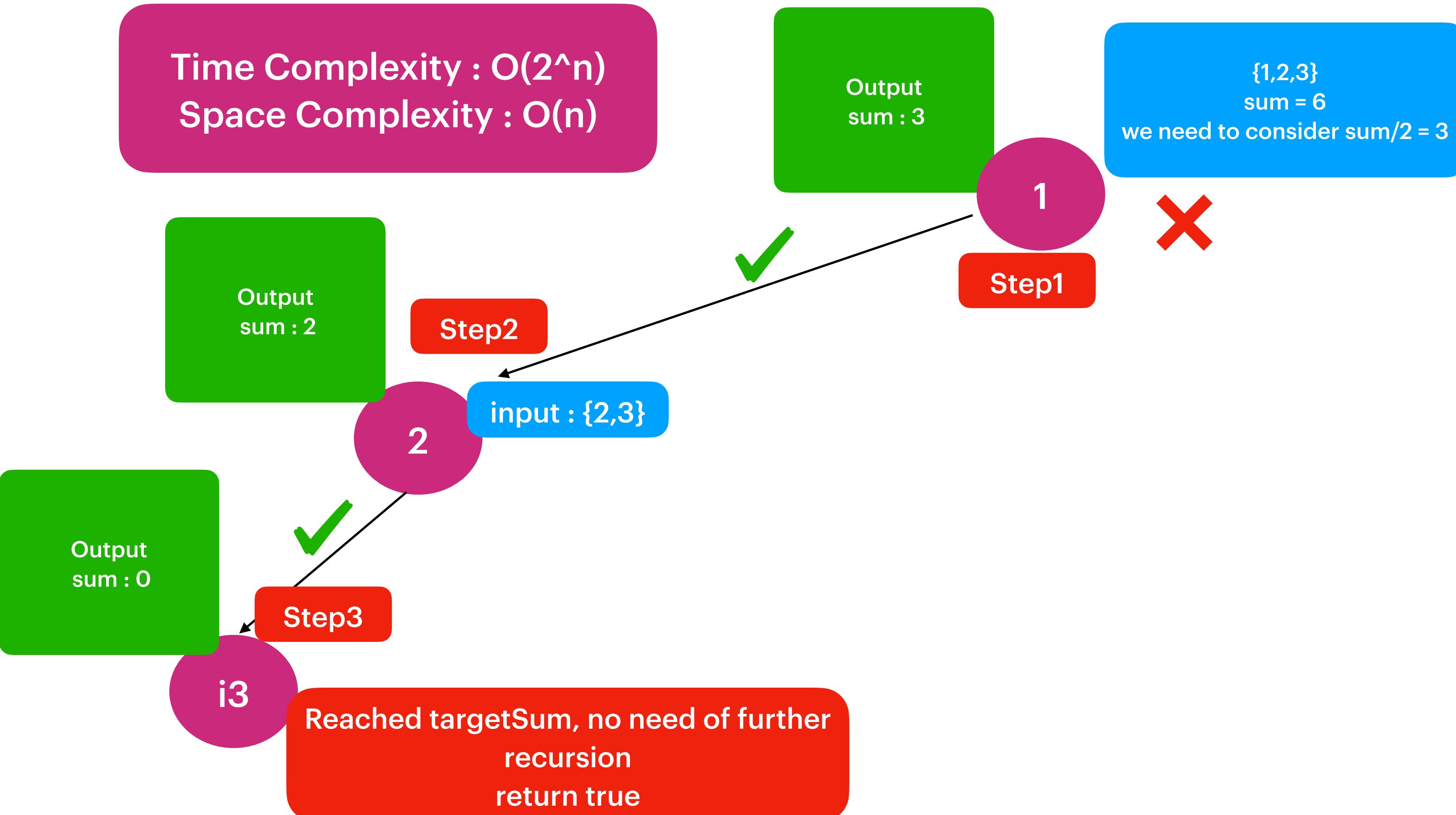
**Input: {1, 2, 3, 4}**

**Output: True**

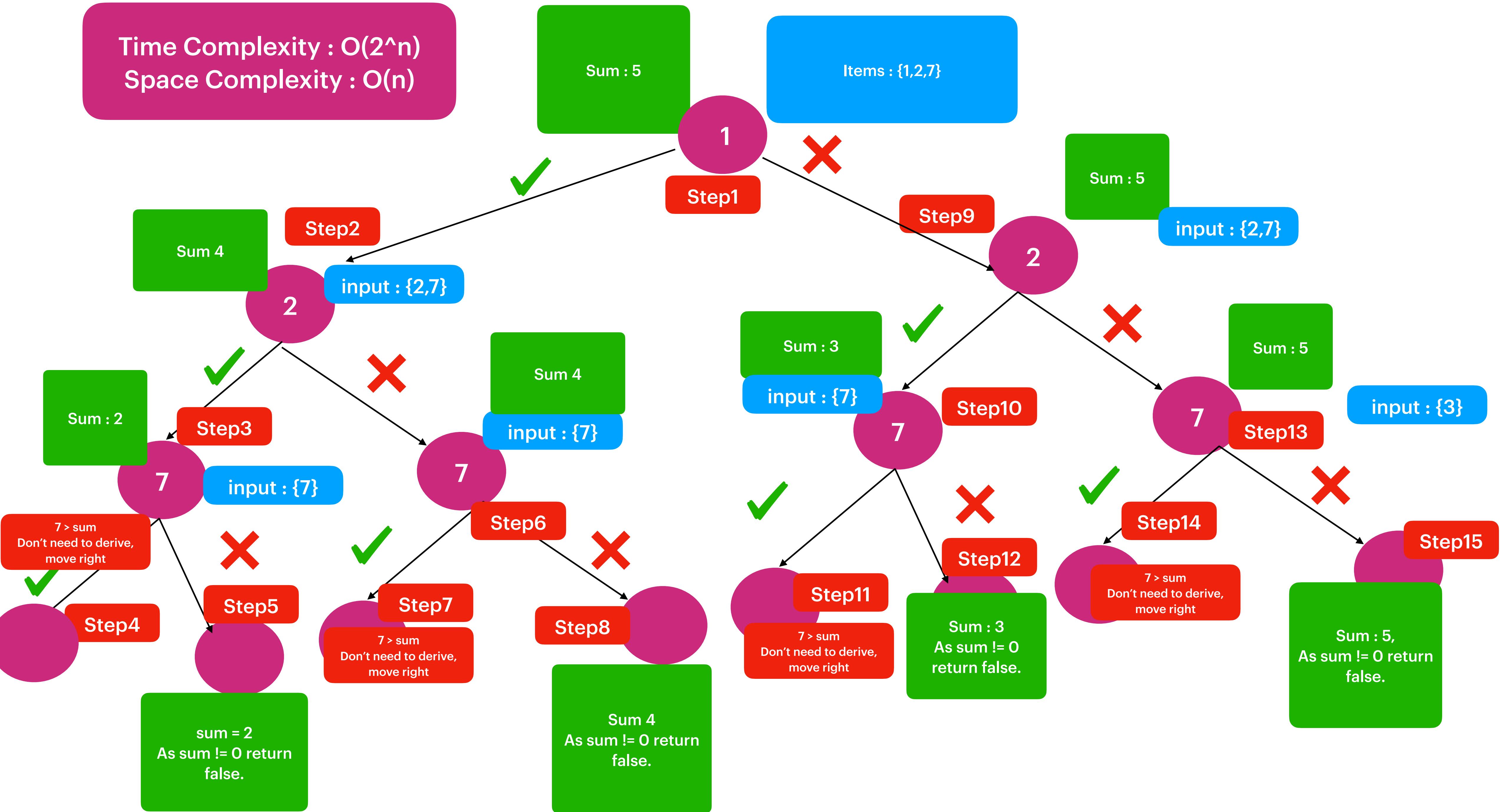
**Explanation:** The given set can be partitioned into two subsets with equal sum: {1, 4} & {2, 3}



Time Complexity :  $O(2^n)$   
Space Complexity :  $O(n)$



Time Complexity :  $O(2^n)$   
Space Complexity :  $O(n)$



We have Items, every Item has weight and profit.

## 0/1 Knapsack

Fixed Capacity

### Minimum Subset Sum Difference to make equal partition:

**Problem Statement :** Given a set of positive numbers, partition the set into two subsets with a minimum difference between their subset sum

Input: {1, 2, 7}

Expected Output : 4

The possible subsets are {1,2} & {7} so we need '4' to make minimal possible two subsets.

Input: {1, 2, 3, 9}

Expected Output : 3

The possible subsets are {1,2,3} & {9} so we need '3' to make minimal possible two subsets.

Input: {2,7,8}

Expected Output : 1

The possible subsets are {2,7} & {8} so we need '1' to make minimal possible two subsets.

Input: {1, 3, 100,4}

Expected Output : 92

The possible subsets are {1,3,4} & {100} so we need '92' to make minimal possible two subsets.

Input: {1, 2,3,4}

Expected Output : 0

We can partition the given array into two subsets

```

int[] arr = {1,2,7}
totalSum = 10
sum = totalSum/2 = 5

```

### Minimum Subset Sum Difference

Tabulation :  $dp[n][sum/2]$   
 TimeComplexity :  $O(n*sum)$   
 SpaceComplexity :  $O(n*sum)$

Expected output {4}

Value	Index
1	0
2	1
7	2

totalSum = 10  
 leftSum = 3  
 rightSum =  $10-3 = 7$   
 expectedMinValue = rightSum - leftSum =  $7-3 = 4$

Target Sum					
0	1	2	3	4	5
T	T	F	F	F	F
T	T	T	T	F	F
T	T	T	T	F	F

leftSum = 3  
 targetSum=3 is possible without including last element.(i.e {1,2} )

If we include 4 in the input array, we could make equal subset sum partition.  
{1,2,4} {7}

Algorithm:

=> Derive tabulation by just following equal subset sum partition Algorithm.

=> Find out what the max subset can be possible with out including last element by moving left to right in n-1 row.

=> The true value talks about possible leftSum, then calculate the rightSum find out the difference.

## Minimum Subset Sum Difference to make equal partition:

Memoization :  $dp[n][sum]$   
TimeComplexity :  $O(n*sum)$   
SpaceComplexity :  $O(n*sum)$

How to approach ?



Observe the input {7,2,8}

Blindly look at all possible subsets with including element & without including an element ? {7,2,8} sum is 17. Lets calculate , includeSum, excludeSum & diff, for each SubSet.

{ } => includeSum = 0 excludeSum = 17 diff = 17

{7} => includeSum = 7 excludeSum = 10 diff = 3

{2} => includeSum = 2 excludeSum = 15 diff = 13

{8} => includeSum = 8 excludeSum = 9 diff = 1

{7,2} => includeSum = 9 excludeSum = 8 diff = 1

{7,8} => includeSum = 15 excludeSum = 2 diff = 13

{2,8} => includeSum = 10 excludeSum = 7 diff = 3

{7,2,8} => includeSum = 17 excludeSum = 0 diff = 17

Got it expected value would be minimum diff i .e 1

Memoization :  $dp[n][sum]$   
TimeComplexity :  $O(n*sum)$   
SpaceComplexity :  $O(n*sum)$

includeSum : 0  
excludeSum : 0  
diff : 0

Items : {1,2,7}  
sum: 10

includeSum : 0  
excludeSum : 1  
diff : 1

input : {2,7}

includeSum : 1  
excludeSum : 0  
diff : 1

Step2

input : {2,7}

2

1

Step1

Step9

2

includeSum : 2  
excludeSum : 1  
diff: 1

input : {7}

Step10

includeSum : 0  
excludeSum : 3  
diff : 3

input : {3}

7

includeSum : 3  
excludeSum : 0  
diff: 3

Step3

input : {7}

7

7

Step6

input : {7}

7

7

Step12

includeSum : 0  
excludeSum : 10  
diff:10

Step15

includeSum : 10  
excludeSum : 0  
diff: 10

Step4

Step5

includeSum : 3  
excludeSum : 7  
diff:4

includeSum : 8  
excludeSum : 2  
diff:6

Step8

includeSum : 1  
excludeSum : 9  
diff: 8

Step11

includeSum : 9  
excludeSum : 1  
diff: 8

Step12

includeSum : 2  
excludeSum : 8  
diff: 6

Step14

includeSum : 7  
excludeSum : 3  
diff: 4

return minimum diff i.e 4

We have Items, every Item has weight and profit.

## 0/1 Knapsack

Fixed Capacity

## Count of Subset Sum

**Problem Statement :** Given a set of positive numbers, find the total number of subsets whose sum is equal to a given number 'S'.

**Input:** {2,3,5}, S=5

**Output:** 2

The given set has '2' subsets whose sum is '5': {2,3}, {5}

**Input:** {1, 1, 2, 3}, S=4

**Output:** 3

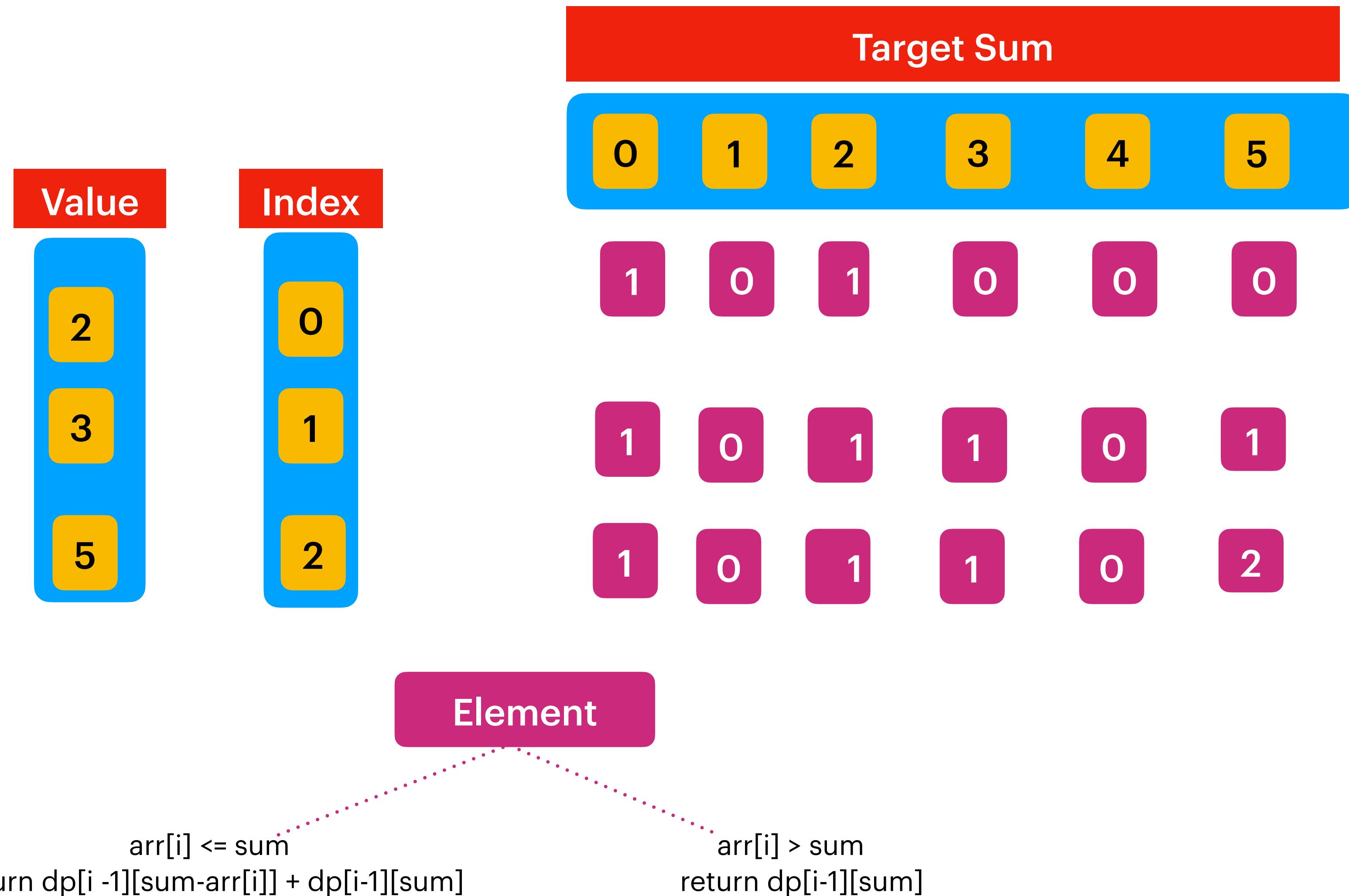
The given set has '3' subsets whose sum is '4': {1, 1, 2}, {1, 3}, {1, 3}

Note that we have two similar sets {1, 3}, because we have two '1' in our input.

```
int[] arr = {2,3,5}  
targetSum = 5  
output : 2
```

### Count of Subset Sum

Tabulation :  $dp[n][sum]$   
TimeComplexity :  $O(n*sum)$   
SpaceComplexity :  $O(n*sum)$



## Count of Subset Sum



How to approach ?



Observe the input {2,3,5} targetSum = 5



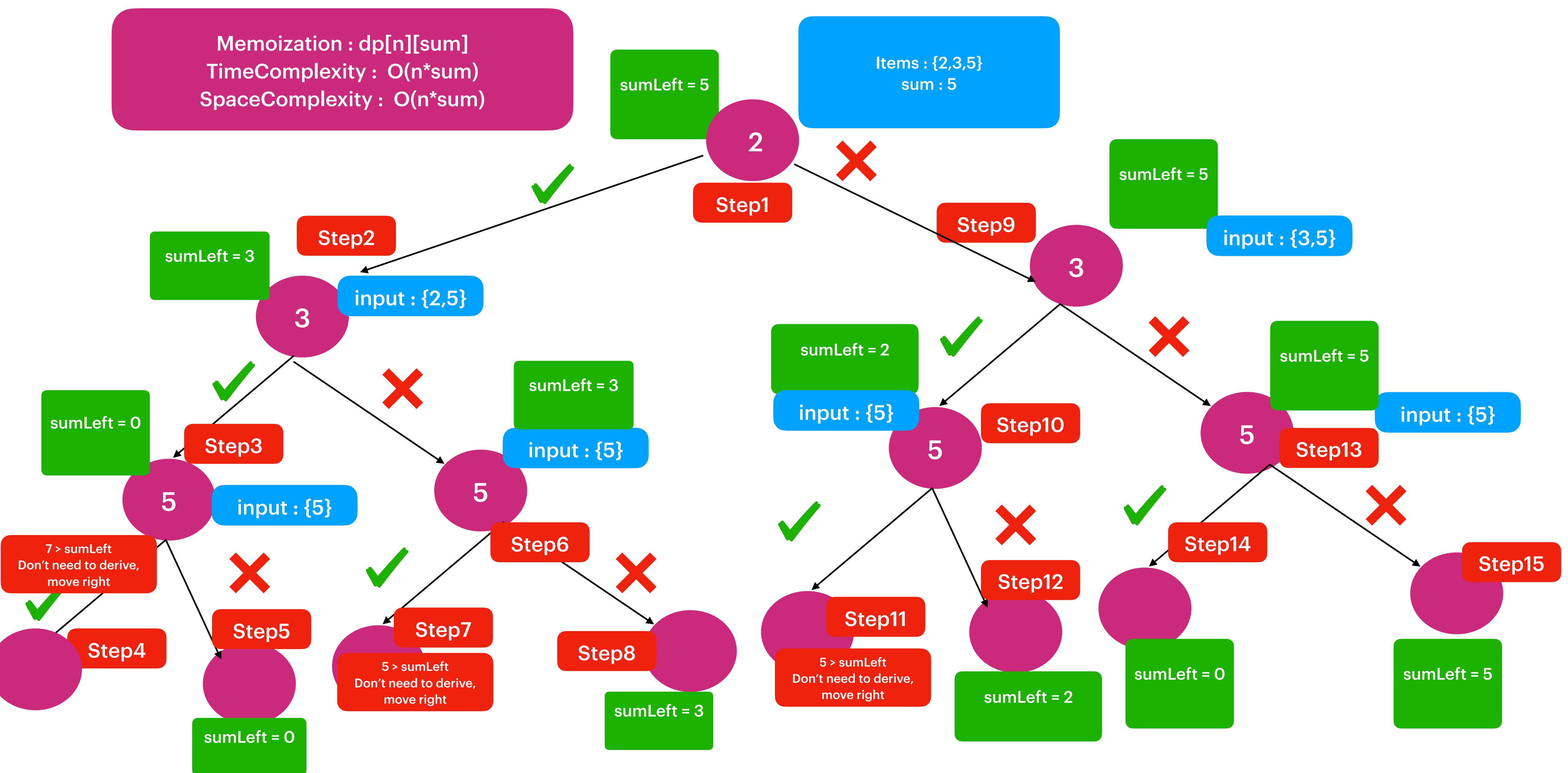
Memoization :  $dp[n][sum]$   
TimeComplexity :  $O(n*sum)$   
SpaceComplexity :  $O(n*sum)$

Blindly look at all possible subsets

{ } => sum = 0  
{2} => sum = 2  
{3} => sum = 3  
{5} => sum = 5  
{2,3} => sum = 5  
{2,5} => sum = 7  
{3,5} => sum = 8  
{2,3,5} => sum = 10

Got it we reached targetSum twice so outPut : 2

**Memoization :  $dp[n][sum]$**   
**TimeComplexity :  $O(n*sum)$**   
**SpaceComplexity :  $O(n*sum)$**



We have Items, every Item has weight and profit.

## Unbounded Knapsack Problems

Fixed Capacity

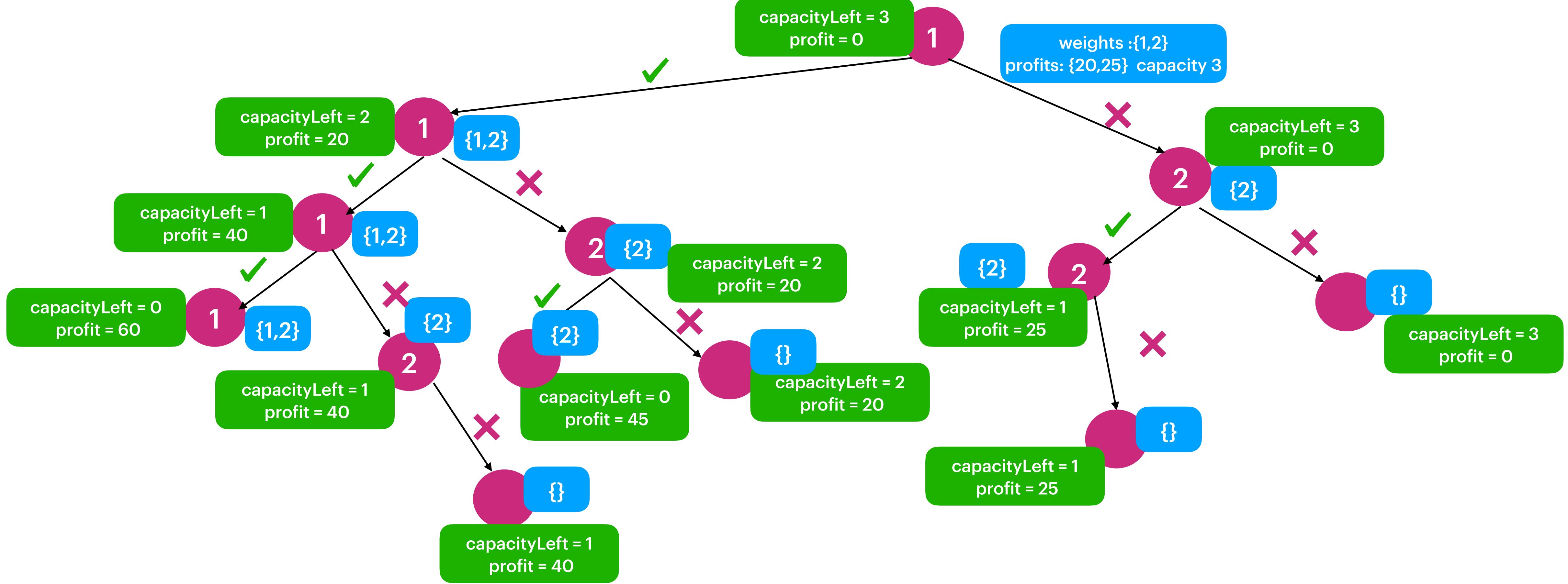
**Problem Statement :** Given item weights and their respective profits, itemWeight = {1,2,3} ItemsProfit = {5,2,4} , Get the max profit. We can have duplicate Items.

Constraints :

The knapsack capacity : 5

Expected Output :

Max Profit we can gain by choosing itemWeights {1,1,1,1,1} & respective profits  
 $\{5,5,5,5,5\} = 25$



Time Complexity :  $2^{n+c}$   
 Space Complexity :  $O(n+c)$

Only difference, we can move to next element only when we exclude item.

## UnBounded Knapsack Problem

weights {1,2,3}  
profits {5,2,4}

capacity : 2

weights[element] <= capacity

Element

weights[element] > capacity

Element adds into the bag

Element can't be included into the bag

calculate profit with including current item

calculate profit without including current item

$\max(\text{profits}[i] + \text{dp}[i][c-\text{weight}[i]], \text{dp}[i-1][c])$

$\text{dp}[i-1][c]$

Get max profit then return

`itemWeight = {1,2,3}`  
`ItemsProfit = {5,2,4},`  
`capacity : 5`

For zero capacity all the items gives 0 profit.

Profits	Weight	Index
5	1	0
2	2	1
4	3	2

Target Sum					
0	1	2	3	4	5
0	5	10	15	20	25
0	5	10	15	20	25
0	5	10	15	20	25

When we are solving for the 1st element.  
i.e at index 0.

We don't have option of comparison so

If weight of the 1st element is  $\leq$  capacity  
then value will be,  $\text{profits}[0] + \text{dp}[0][\text{c-weight}[0]]$ .

If weight of the 1st element is  $>$  capacity then value is 0.

$\leq$  capacity

$>$  capacity

Element weight

Max(Including element , excluding element)  
 $\max(\text{profits}[i] + \text{dp}[i][\text{c-weight}[i]], \text{dp}[i-1][\text{c}])$

Exclude Element  
 $\text{dp}[i-1][\text{c}]$

We have Items, every Item has weight and profit.

Unbounded Knapsack Problems

Fixed Capacity

### Rod Cutting :

Given a rod of length 'n', we are asked to cut the rod and sell the pieces in a way that will maximize the profit. We are also given the price of every piece of length 'i' where ' $1 \leq i \leq n$ ' .

Example:

Lengths: [1, 2, 3, 4, 5]

Prices:[2, 6, 7, 10, 13]

Rod Length: 5

Output : 14

{1,2,2} max profit can be possible.

Lets see all possible positive options :::

Lengths: [1, 2, 3, 4, 5]

Prices: [2, 6, 7, 10, 13]

Rod Length: 5

=> {1,1,1,1,1} =>  $5 * 2 = 10$  RS

=> {1,1,1,2} =>  $3 * 2 + 6 = 12$  RS

=> {1,1,3} =>  $2 * 2 + 7 = 11$

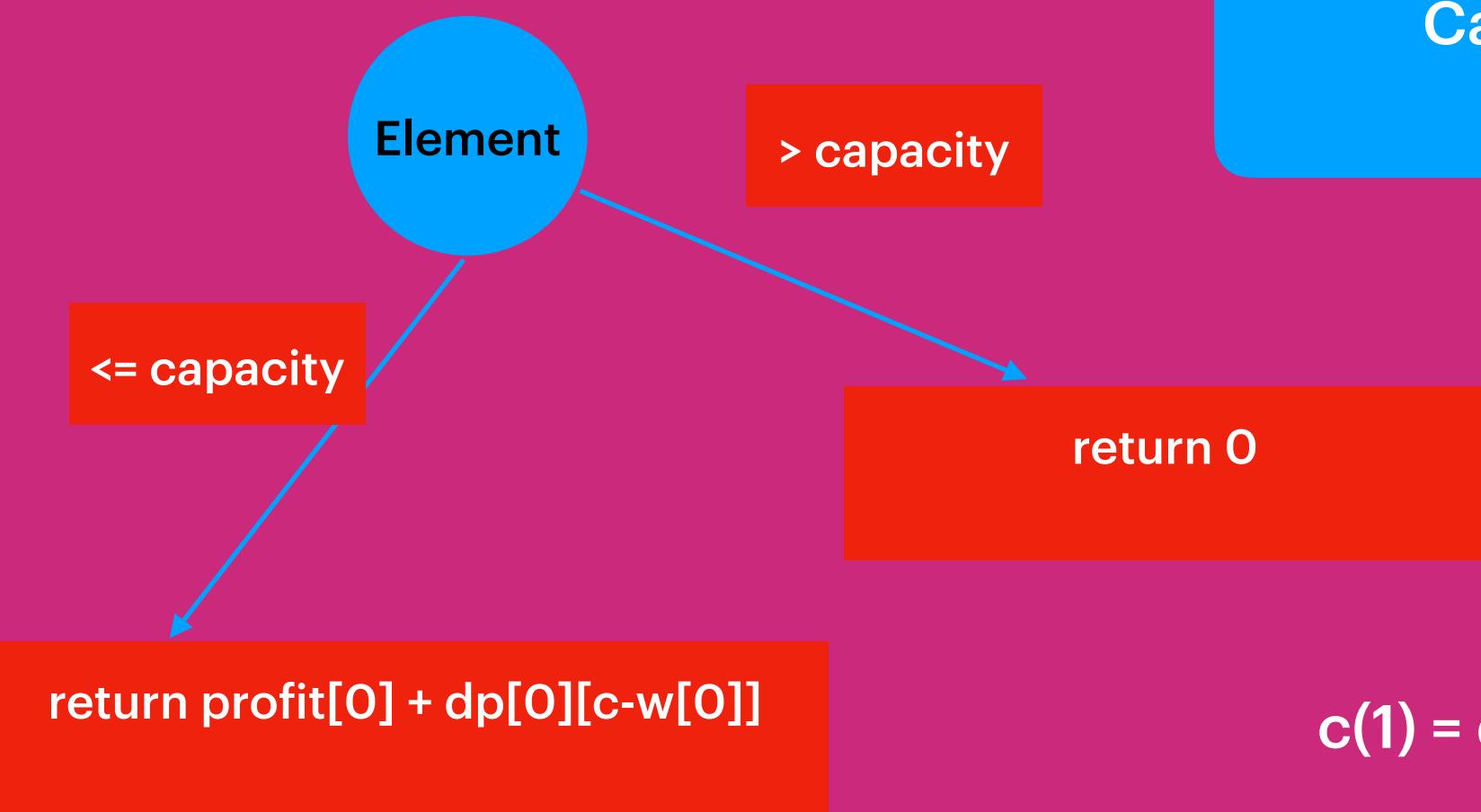
=> {1,4} =>  $2 + 10 = 12$

=> {1,2,2} =>  $1 * 2 + 2 * 6 = 14$  RS

=> {2,3} =>  $1 * 6 + 1 * 7 = 13$  RS

=> {5} =>  $1 * 13 = 13$  RS

So Max Profit we got is {1,2,2} => 14



Lengths: [1, 2, 3, 4, 5]  
 Prices: [2, 6, 7, 10, 13]  
 Rod Length: 5

**Calculating subproblem with element 1 :**  
 Index = 0, w{1} p{2} , Rod Length: 5

$$c(5) = dp[0][5] = \{1,1,1,1,1\} \Rightarrow 5 * 2 = 10$$

Lets see how it works !!!!

$$c(0) = dp[0][0] = w(0) > c(0) = 0$$

$$\begin{aligned} c(1) &= dp[0][1] = p[0] + dp[0][1-1] = p[0] + dp[0][0] \\ &= 2 + c(0) = 2 + 0 = 2 \end{aligned}$$

$$\begin{aligned} c(2) &= dp[0][2] = p[0] + dp[0][2-1] = p[0] + dp[0][1] \\ &= p[0] + c(1) = 2 + 2 = 4 \end{aligned}$$

$$\begin{aligned} c(3) &= dp[0][3] = p[0] + dp[0][3-1] = p[0] + dp[0][2] = p[0] + c[2] \\ &= 2 + 4 = 6 \end{aligned}$$

$$c(4) = p[0] + c(3) = 2 + 6 = 8$$

$$c(5) = p[0] + c(4) = 2 + 8 = 10$$

**Calculating subproblem with element 1 :**

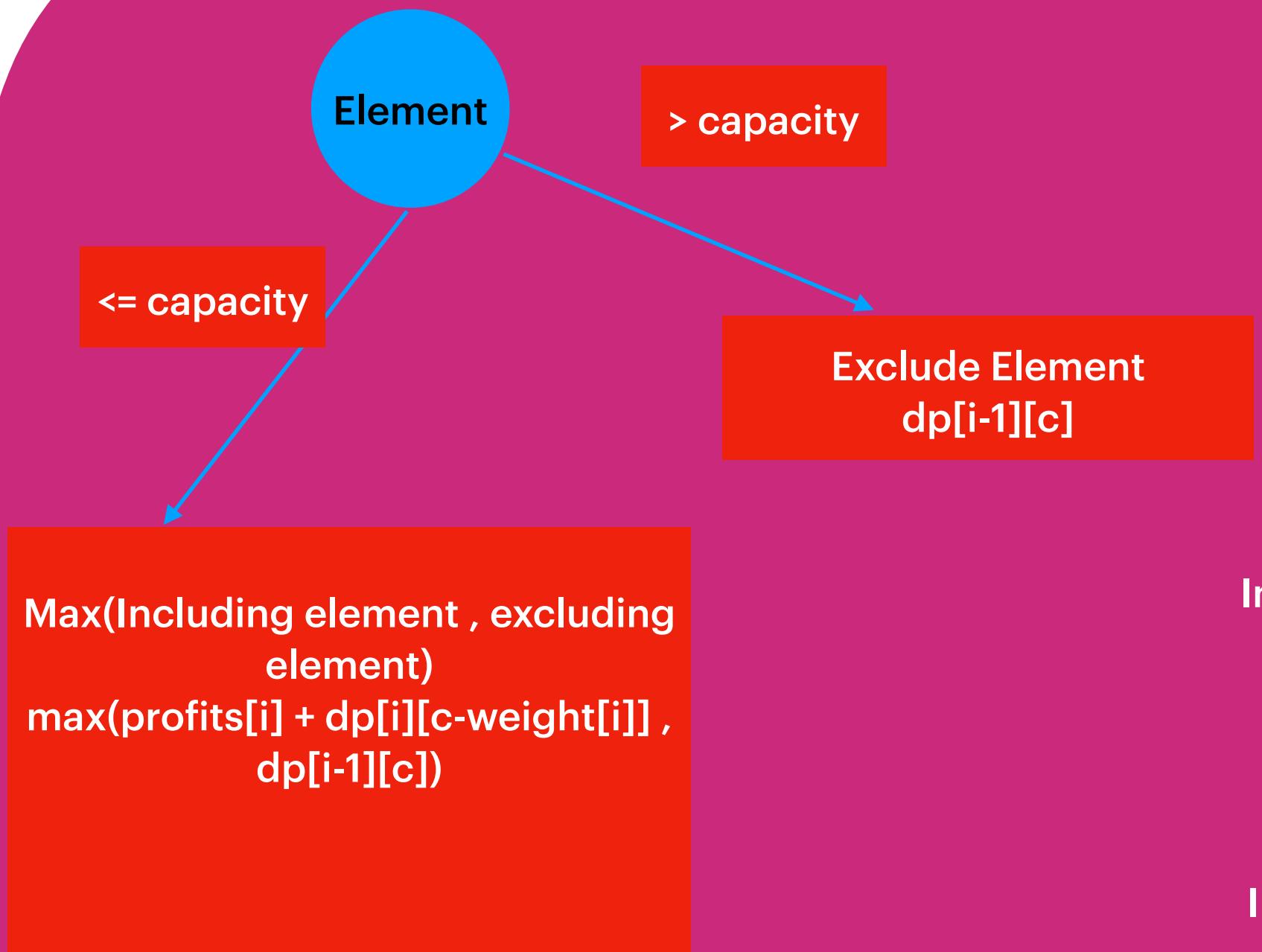
When we consider only one element, we should see either we can include or exclude the element.

If weight of element > capacity  
 then profit = 0

Otherwise

profit = currentElementProfit profitOfCapacityLeft  
 $= \text{profit}[0] + dp[0][\text{capacity} - w[0]]$

Calculating subproblem with elements 2  
Index = 1, w{1,2} p{2,6}, Rod Length i.e capacity : 5



$c(0) = dp[1][0] = w(2) > c(0) = 0$   
 $c(1) = dp[1][1] = w(2) > c(1) = dp[0][1] = 2$   
 $c(2) = dp[1][2] = w(2) \leq c(2) = 6$   
 Exclude Element =  $dp[0][2] = 4$   
 Include Element =  $p[1] + dp[1][2-2] = p[1] + dp[1][0] = 6 + 0 = 6$   
 Max(Including Element, Excluding Element)  
 Max(6,4) = 6

$c(3) = dp[1][3] = w(2) \leq c(3) =$   
 Exclude Element =  $dp[0][3] = 6$   
 Include Element =  $p[1] + dp[1][3-2] = p[1] + dp[1][1] = 6 + 2 = 8$   
 Max(Including Element, Excluding Element)  
 Max(8,6) = 8

$c(4) = dp[1][4] = w(2) \leq c(4) =$   
 Exclude Element =  $dp[0][4] = 8$   
 Include Element =  $p[1] + dp[1][4-2] = p[1] + dp[1][2] = 6 + 6 = 12$   
 Max(Including Element, Excluding Element)  
 Max(12,8) = 12

$c(5) = dp[1][5] = w(2) \leq c(5) =$   
 Exclude Element =  $dp[0][5] = 10$   
 Include Element =  $p[1] + dp[1][5-2] = p[1] + dp[1][3] = 6 + 8 = 14$   
 Max(Including Element, Excluding Element)  
 Max(14,10) = 14

Lengths: [1, 2, 3, 4, 5]  
 Prices: [2, 6, 7, 10, 13]  
 Rod Length: 5

For zero capacity all the items gives 0 profit.

Prices	Length	Index
2	1	0
6	2	1
7	3	2
10	4	3
13	5	4

Target Sum					
0	1	2	3	4	5
0	2	4	6	8	10
0	2	6	8	12	14
0	2	6	8	12	14
0	2	6	8	12	14
0	2	6	8	12	14

Element weight

<= capacity

> capacity

Max(Including element , excluding element)  
 $\max(\text{profits}[i] + \text{dp}[i][\text{c-weight}[i]], \text{dp}[i-1][\text{c}])$

Exclude Element  
 $\text{dp}[i-1][\text{c}]$

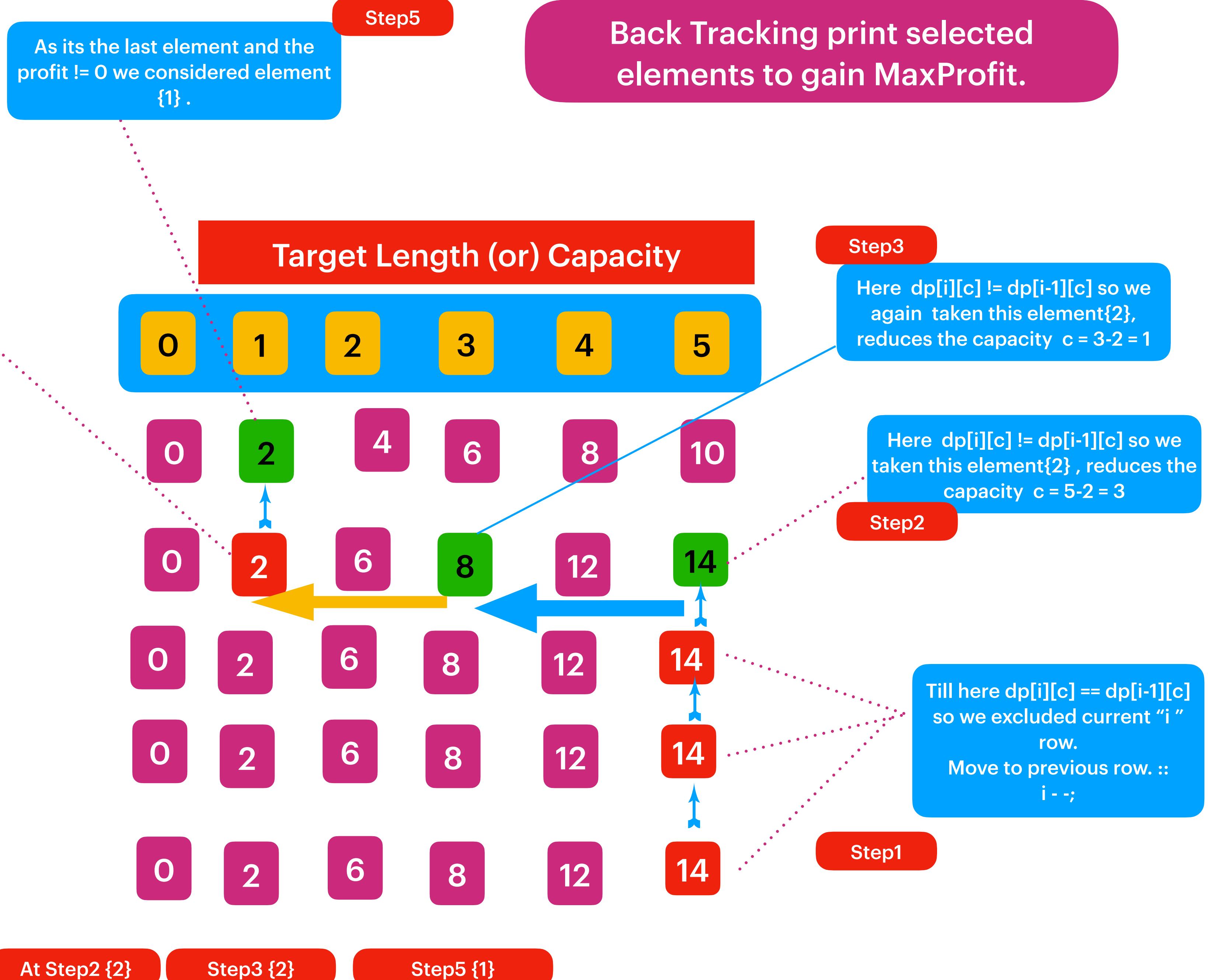
When we are solving for the 1st element.  
 i.e at index 0.

We don't have option of comparison so

If weight of the 1st element is  $\leq$  capacity  
 then value will be,  
 $\text{profits}[0] + \text{dp}[0][\text{c-weight}[0]]$ .

If weight of the 1st element is  $>$  capacity then value is 0.

Prices	Length	Index
2	1	0
6	2	1
7	3	2
10	4	3
13	5	4



We have Items, every Item has weight and profit.

Unbounded Knapsack Problems

Fixed Capacity

### Min Coin Change: (ATM)

Given an infinite supply of 'n' coin denominations and a total money amount, we are asked to find the minimum number of coins needed to make up that amount.

Denominations: {1,2,5,20}

Total amount: 5

Output: 1

Explanation: We need a minimum of 1 coin i.e {5} to make a total of '1'

For the first row , value=  $1+dp[0][c-w[0]]$

For the first row value = 0

weight[i] <= capacity

Target

weight[i] > capacity

Coins

Index

1
2
5
20

0
1
2
3

weight[i] <= capacity

$\min(1+dp[i][c-w[i]], dp[i-1][c])$

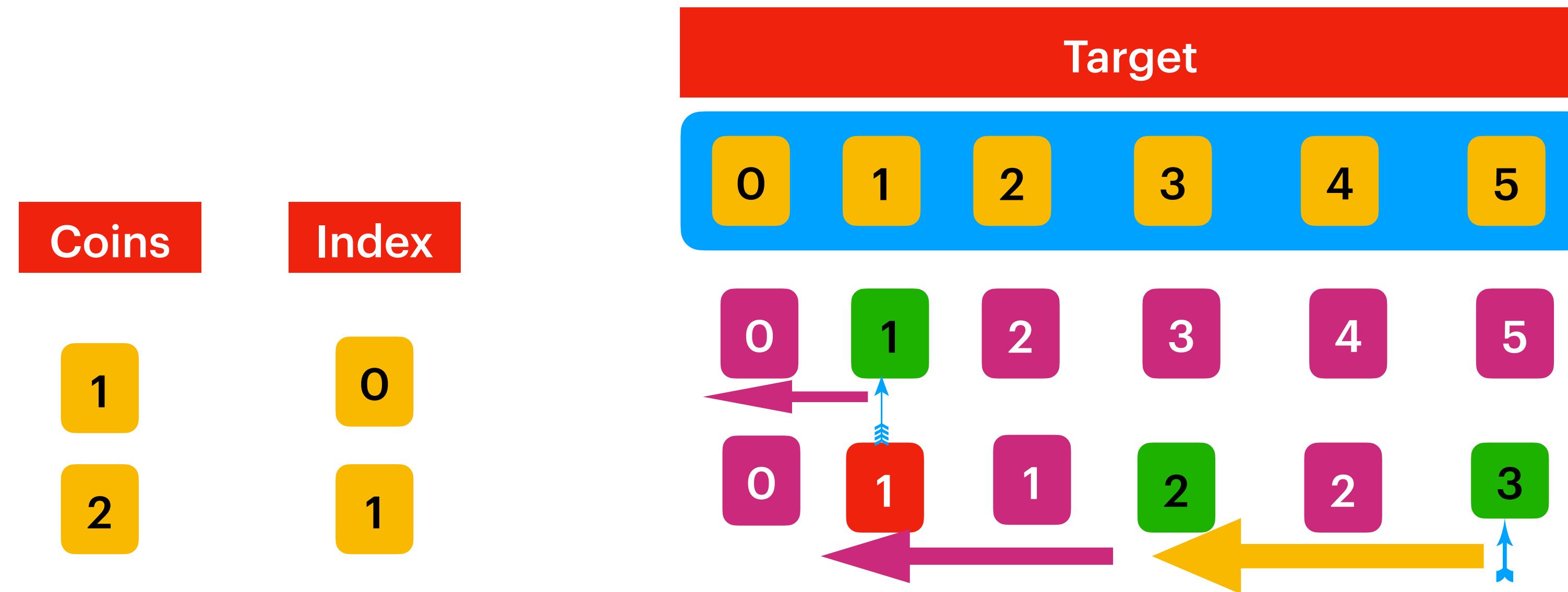
weight[i] > capacity

$dp[i-1][c]$

## BackTracking

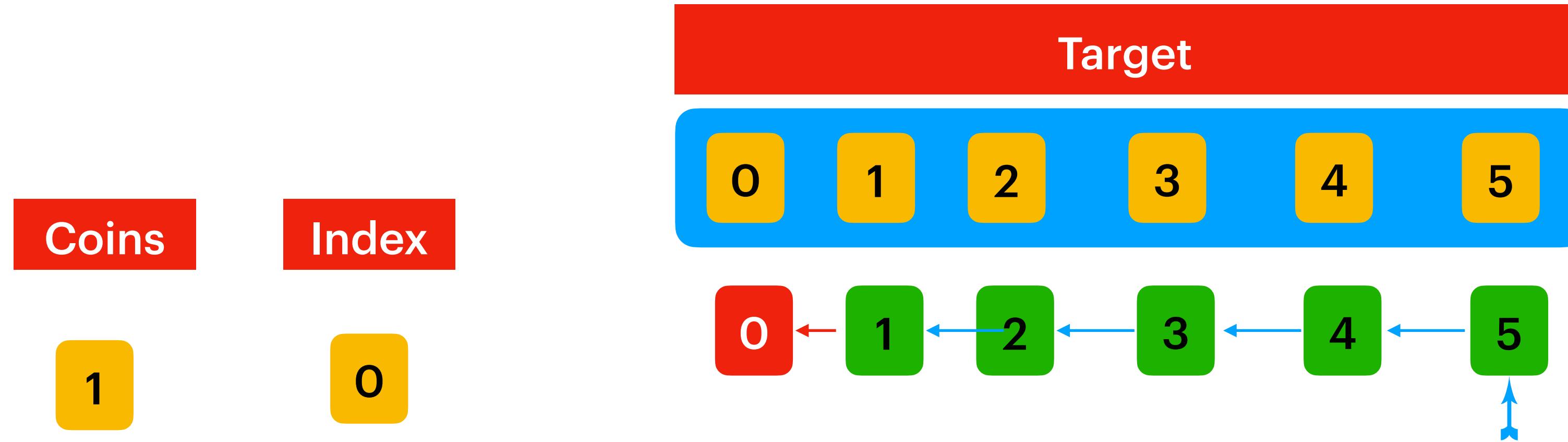


## BackTracking With two elements for capacity 5



So that possible minimal coins {2,2,1}

## BackTracking With 1 elements for capacity 5



So that possible minimal coins {1,1,1,1,1}

We have Items, every Item has weight and profit.

Unbounded Knapsack Problems

Fixed Capacity

Coin Change

Given an infinite supply of 'n' coin denominations and a total money amount, we are asked to find the total number of distinct ways to make up that amount.

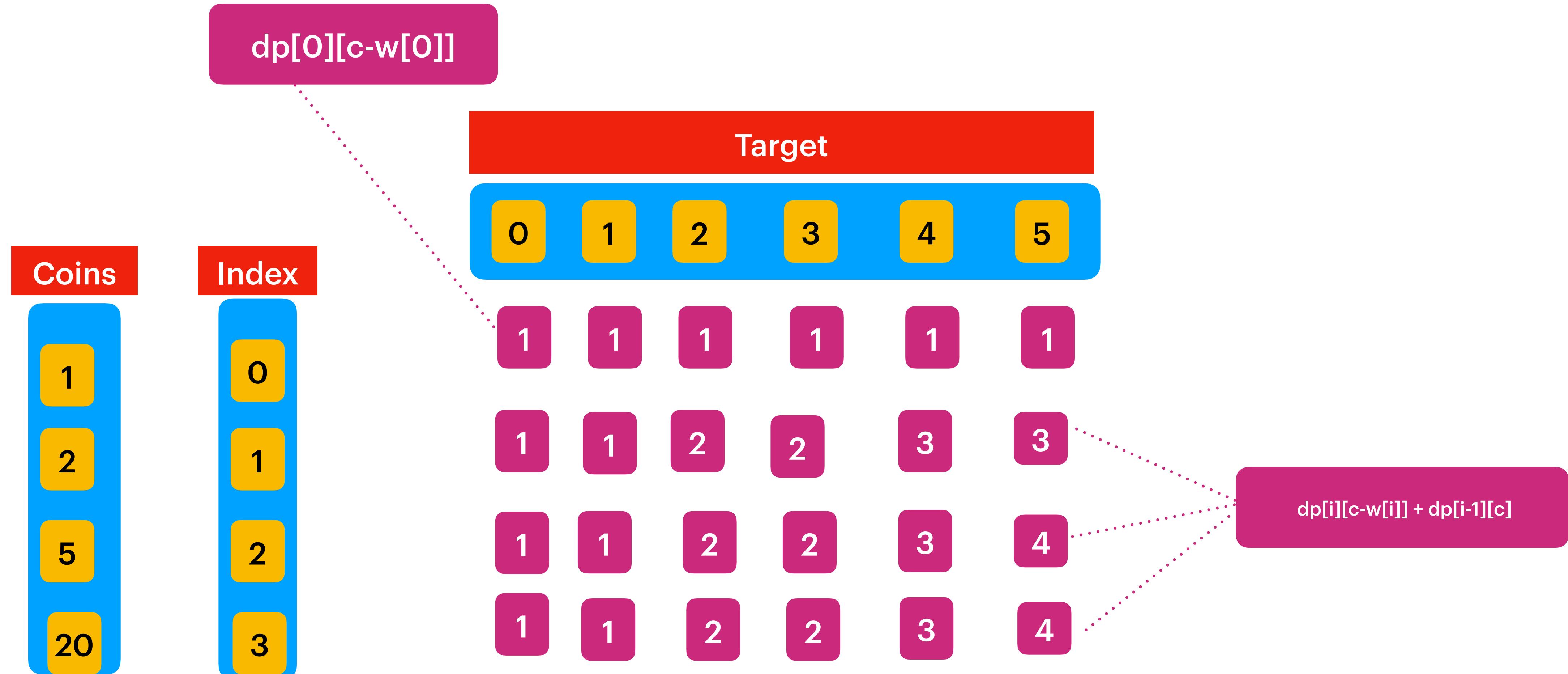
Denominations: {1,2,5,20}

Total amount: 5

Output: 4

Explanation: There are 4 ways to make the change for '5', here are those ways:

1. {1,1,1,1,1}
2. {1,1,1,2}
3. {1,2,2}
4. {5}



## Fibonacci Pattern



**Write a function to calculate the nth Fibonacci number.**

Fibonacci numbers are a series of numbers in which each number is the sum of the two preceding numbers.

First few Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8 .....

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2), \text{ for } n > 1$$

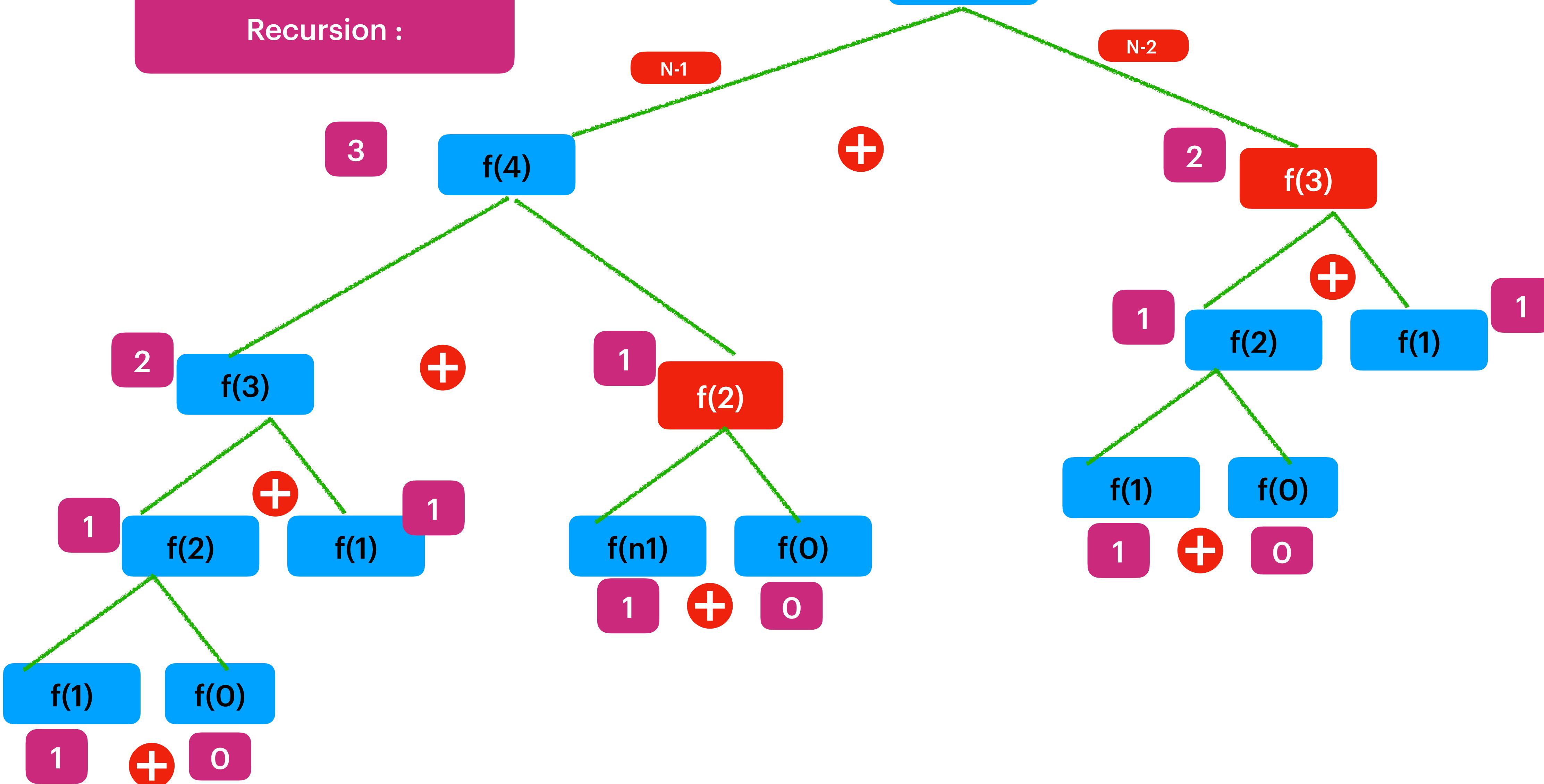
Given that:  $\text{Fib}(0) = 0$ , and  $\text{Fib}(1) = 1$

$f(5)$

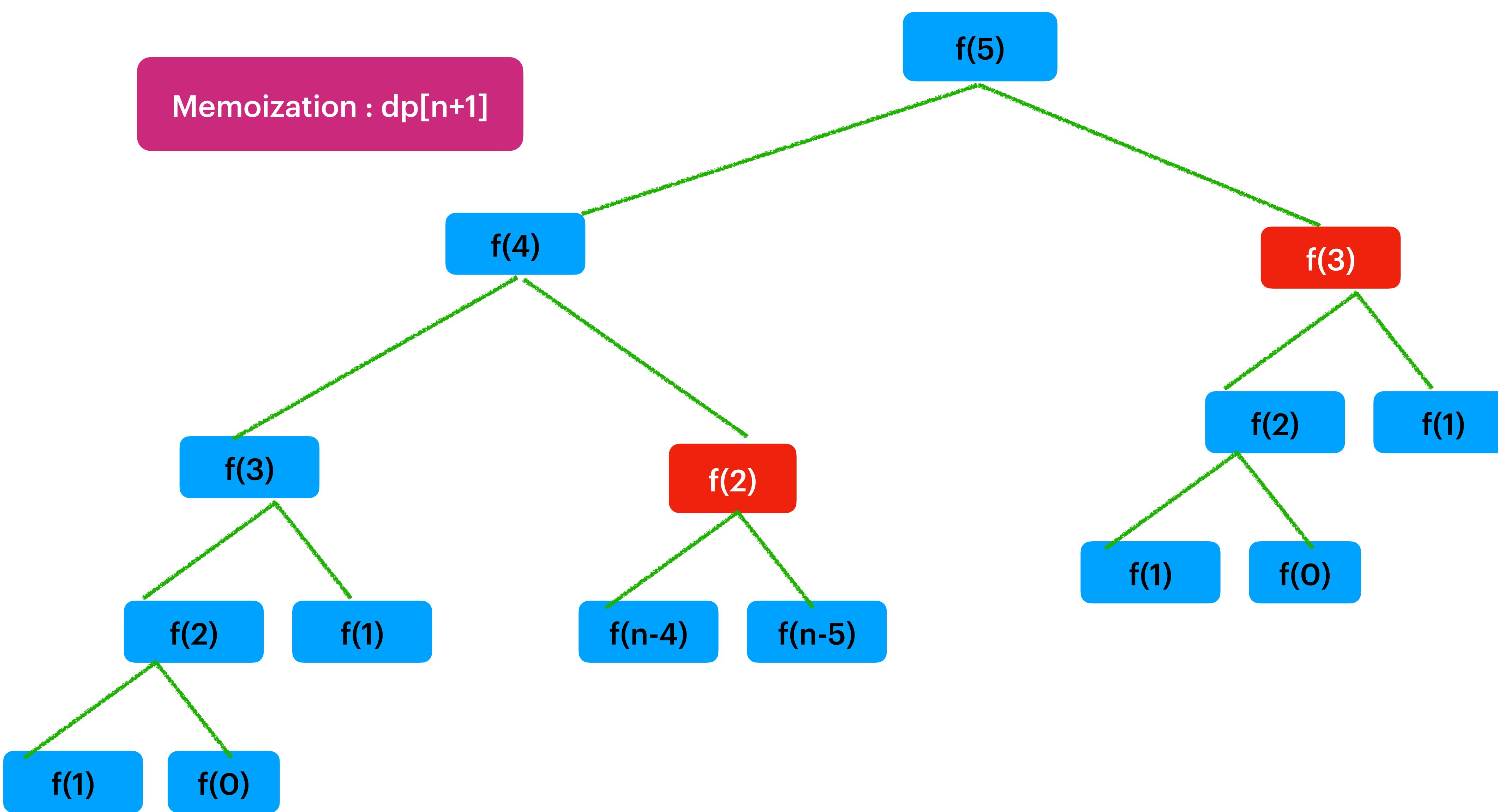
5

Fibonacci numbers 1,1,2,3,5,8,13,21 ...

Recursion :



Memoization :  $dp[n+1]$



**Tabulation  $dp[n+1]$**

$dp[0] = 0$

$dp[1] = 1$

$dp[2] = dp[1] + dp[0] = 0 + 1 = 1$

$dp[n] = dp[n-1] + dp[n-2];$

## Fibonacci Pattern



### Staircase:

Given a stair with 'n' steps, implement a method to count how many possible ways are there to reach the top of the staircase, given that, at every step you can either take 1 step, 2 steps, or 3 steps.

Number of stairs (n) : 3  
Number of ways = 4

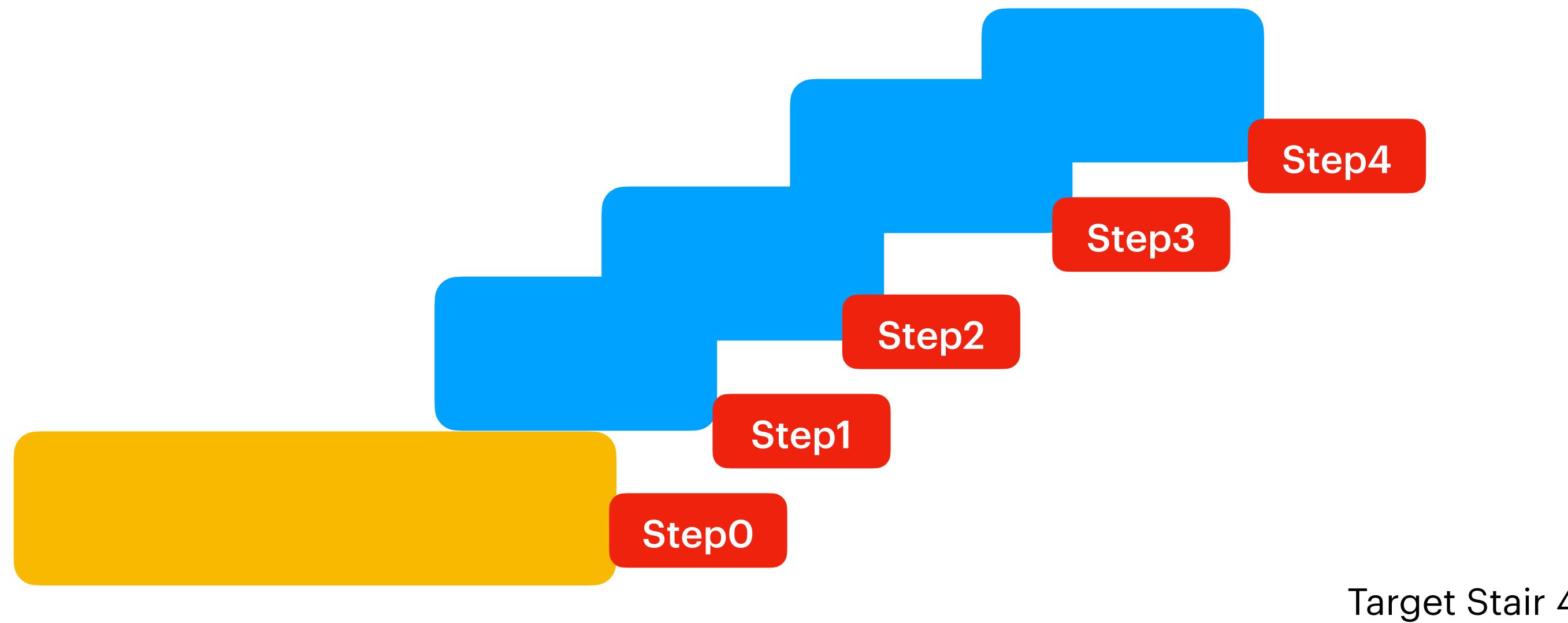
Explanation: Following are the four ways we can climb : {1,1,1}, {1,2}, {2,1}, {3}

Number of stairs (n) : 4  
Number of ways = 7

Explanation: Following are the seven ways we can climb : {1,1,1,1}, {1,1,2}, {1,2,1}, {2,1,1},  
{2,2}, {1,3}, {3,1}

Target is 3 : either you can  
choose step1 or 2 or 3

{1,1,1} ,{1,2}, {2,1} {3} => 4Ways



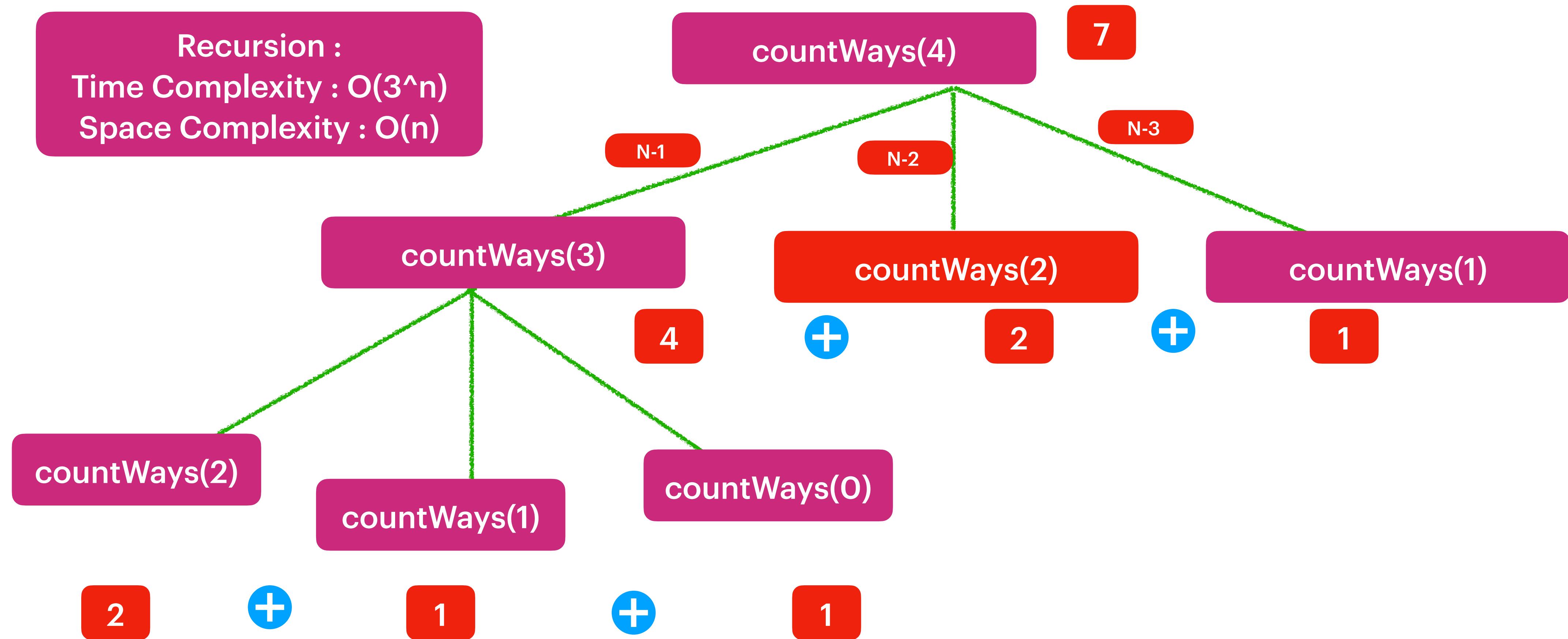
{1,1,1,1} , {1,1,2} , {1,2,1} , {1,3},{2,1,1},{2,2},{3,1} =7

$$2^{-4} = 1/4$$
$$2^{-1} = 1/2$$

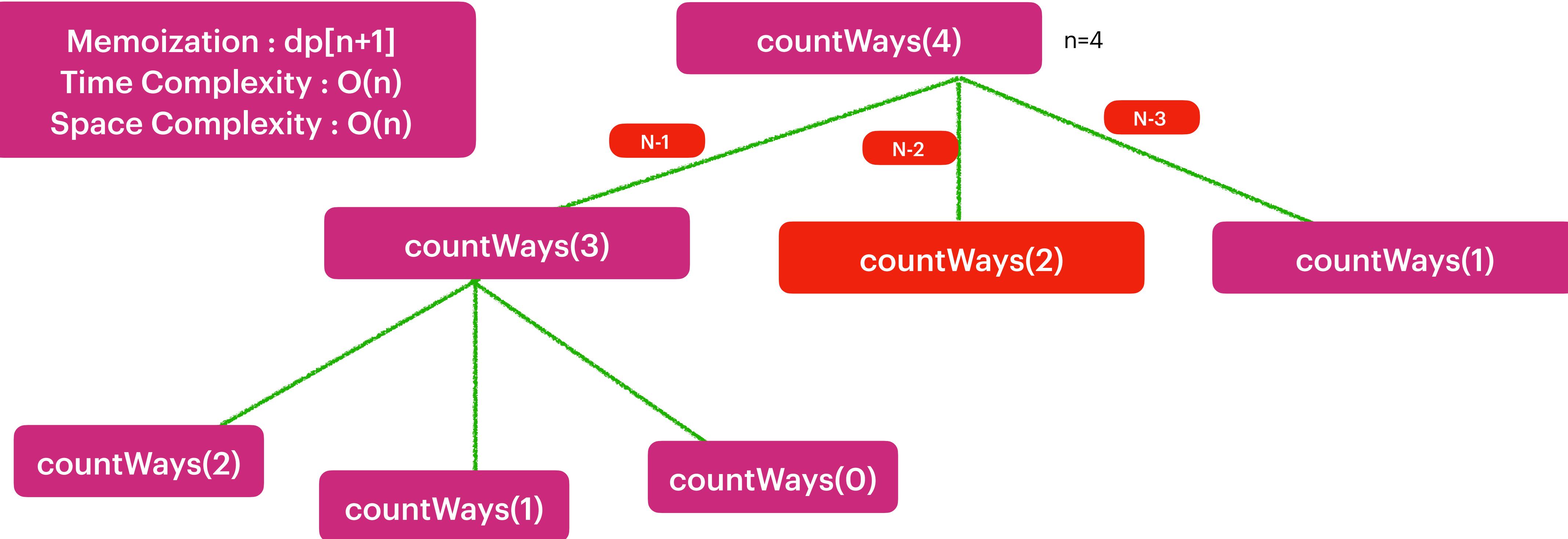
$$2^0 = 1$$
$$2^1 = 2$$
$$2^2 = 4$$
$$2^3 = 8$$
$$\dots 16$$
$$\dots 32$$
$$\dots 64$$

$$0! = 1$$
$$1! = 1$$
$$2! = 2$$
$$3! = 6$$
$$4! = 24$$
$$5! = 120$$

**Recursion :**  
**Time Complexity :**  $O(3^n)$   
**Space Complexity :**  $O(n)$



**Memoization :  $dp[n+1]$**   
**Time Complexity :  $O(n)$**   
**Space Complexity :  $O(n)$**



**Tabulation : dp[n+1]**

**Time Complexity : O(n)**

**SpaceComplexity : O(n)**

**dp[0] = 1; // reach target 0 , {}**

**dp[1] = 1; // reach target 1, {1}**

**dp[2] = 2; // reach target 2 , {1,1} {2}**

**dp[3] = dp[2] + dp[1] + dp[0] = 2+1+1 = 4**

**dp[n] = dp[n-1] + dp[n-2] + dp[n-3];**

$$10 * 2 = 20 \Rightarrow 10 + 10 = 20$$

$$10 * 3 = 30 \Rightarrow 10 + 10 + 10 = 30$$

$$x - 5 = 0$$

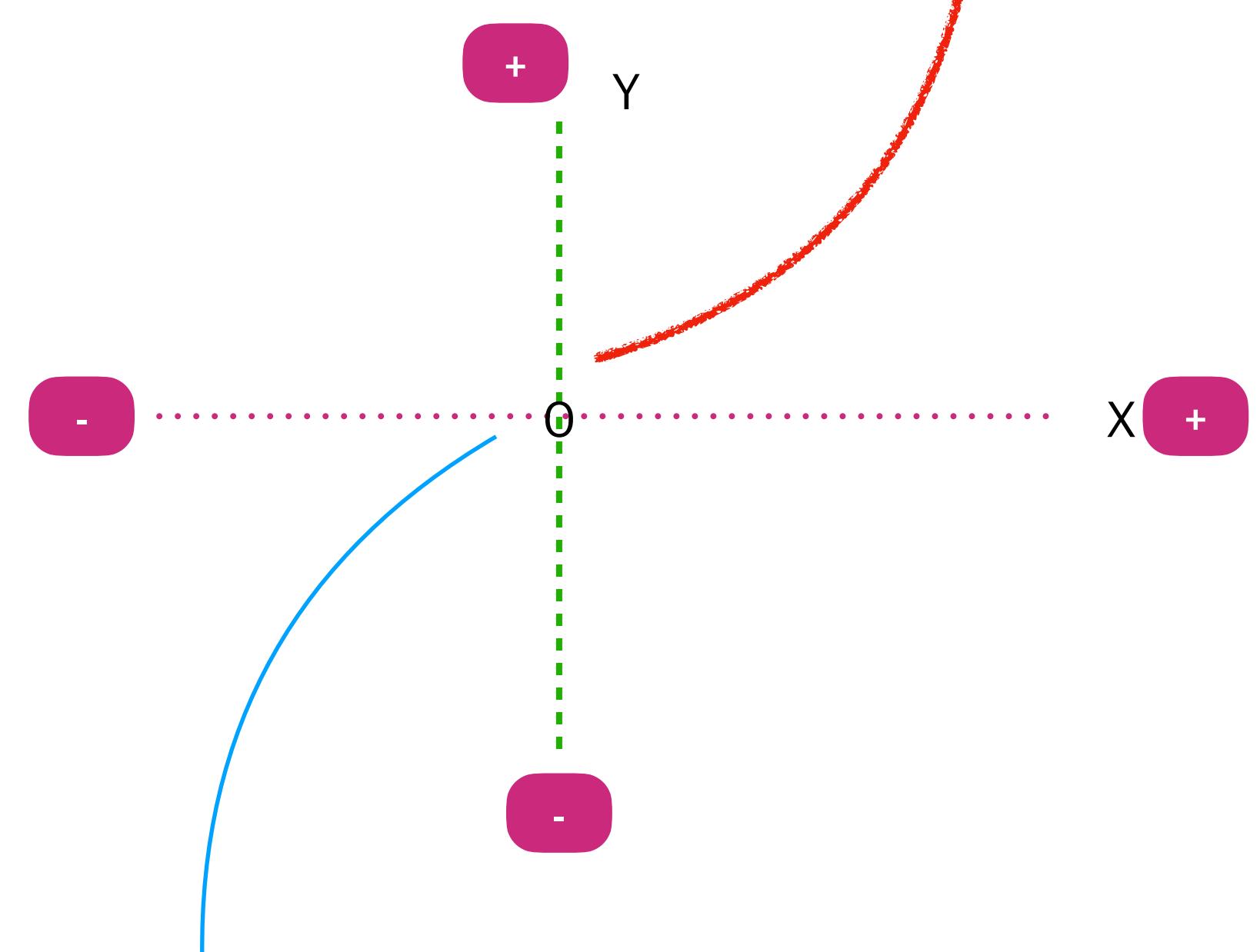
When  $x = 5$

$$5 - 5 = 0$$

$$\begin{aligned} 5/5 &= 1 \\ 5 - 5 &= 0 \end{aligned}$$

$$\begin{aligned} x - 0 &= 0 \\ 0 - 0 &= 0 \\ 0/0 &=? \end{aligned}$$

$$\begin{aligned} 10/2 &= 5 \Rightarrow :: 10 - 2-2-2-2-2 = 0 \\ 10/5 &= 2 \Rightarrow :: 10 - 5-5 = 0 \end{aligned}$$



$$1/0 = \text{Infinity} = 2/0$$

$$1/0 = 2/0$$

$$1 = 2$$

$$c-x = 0$$

$$c/x = 1$$

$$c = x = \text{value}$$

$$C = 1 \quad x = 1$$

$$1/1 = 1$$

$$1-1 = 0$$

$$1/0 = ?$$

$$1/-0.1 = -10$$

$$1/-0.01 = -100$$

$$1/-0.001 = -1000$$

$$1/-0.0001 = -10000$$

....

..... Infinity

$$1/0 = \text{Infinity}$$

$$1/0 = ?$$

$$1/0.1 = 10$$

$$1/0.01 = 100$$

$$1/0.001 = 1000$$

$$1/0.0001 = 10000$$

....

.....

..... Infinity

$$1/0 = \text{Infinity}$$

$$2/0 = ?$$

$$2/0.1 = 20$$

$$2/0.01 = 200$$

$$2/0.001 = 2000$$

$$2/0.0001 = 20000$$

....

.....

..... Infinity

$$2/0 = \text{Infinity}$$

## Fibonacci Pattern



## House thief

There are n houses built in a line. A thief wants to steal the maximum possible money from these houses. The only restriction the thief has is that he can't steal from two consecutive houses, as that would alert the security system. How should the thief maximize his stealing?

### Problem Statement#

Given a number array representing the wealth of n houses, determine the maximum amount of money the thief can steal without alerting the security system

Input: {2, 5, 1}

Output: 5

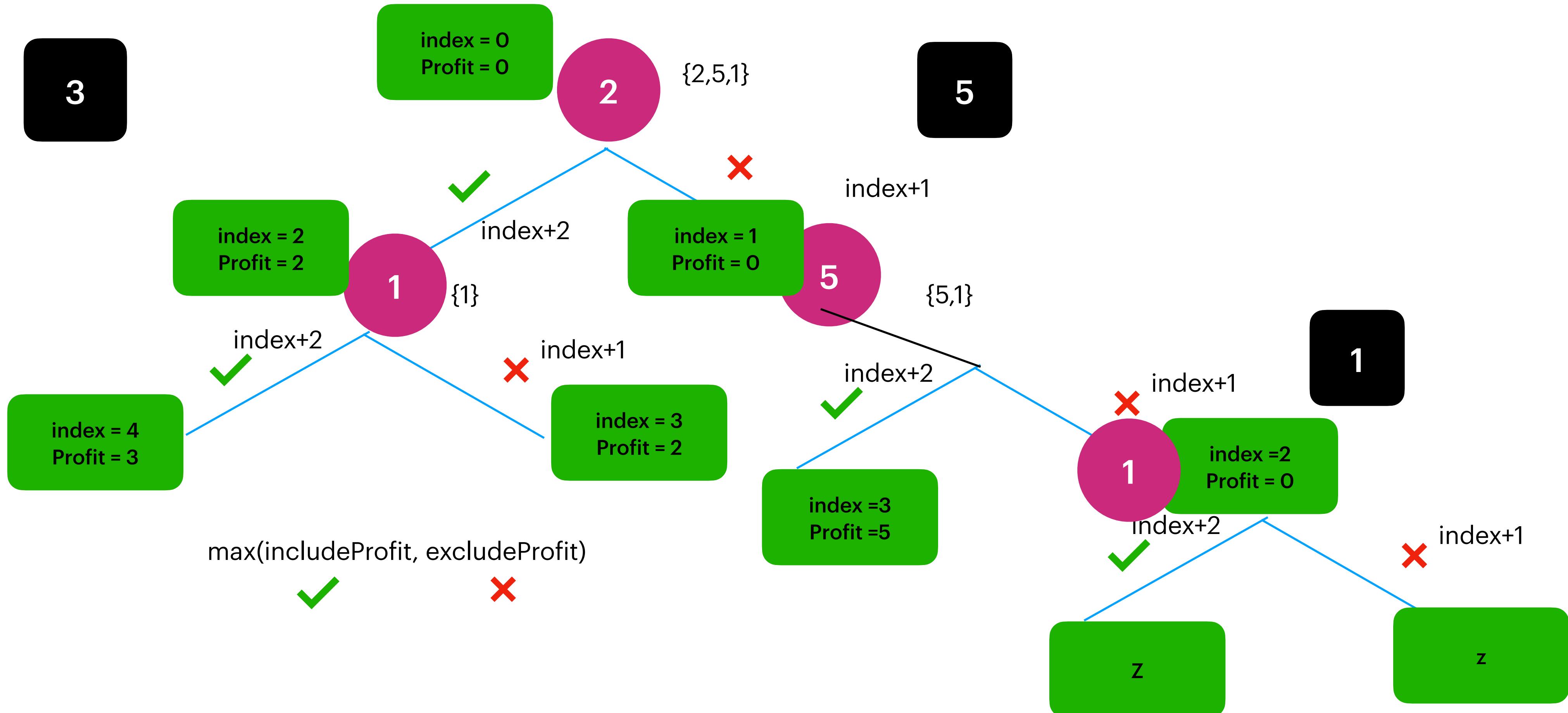
Explanation: The thief should steal from house 5

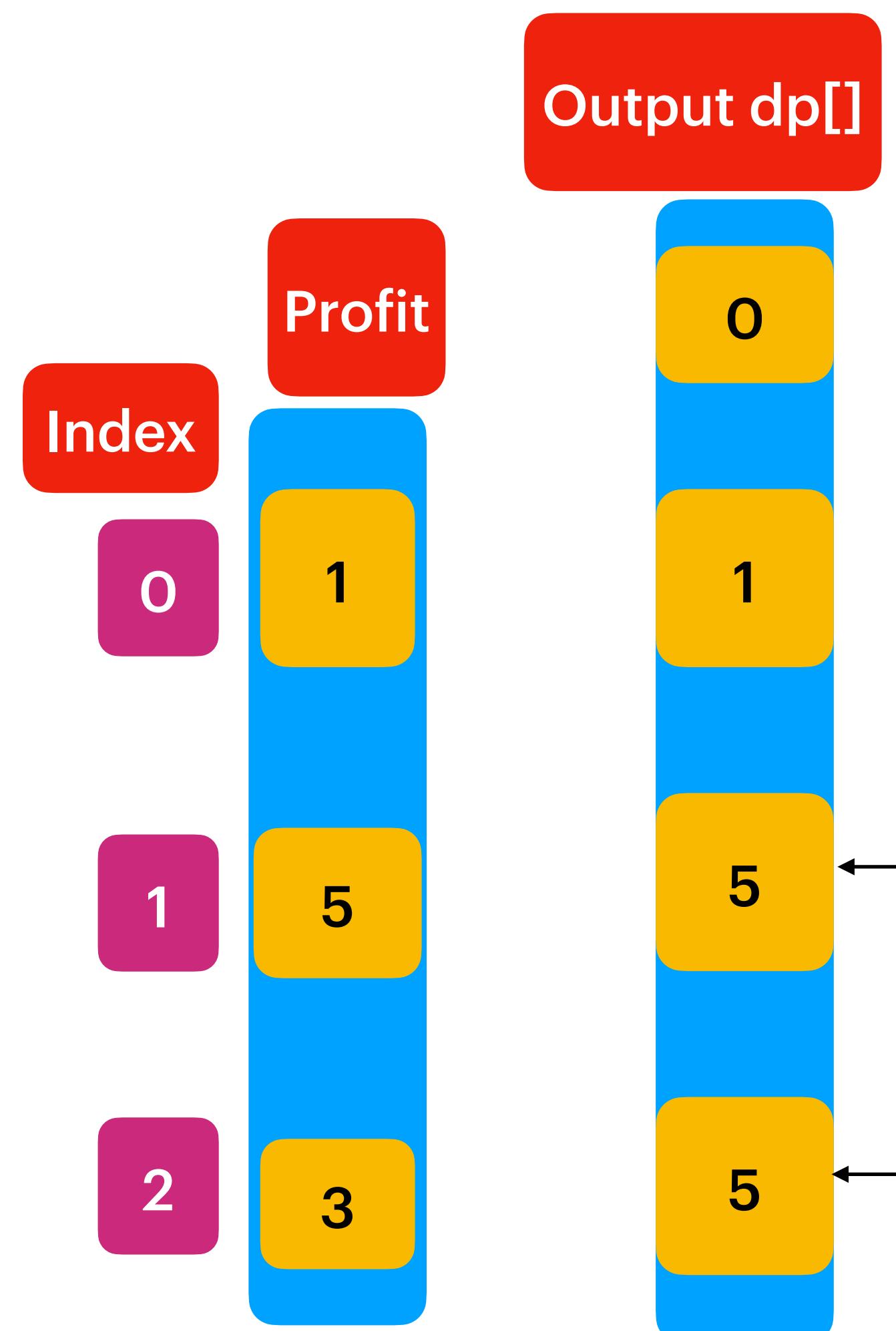
Input: {2, 10, 14, 8, 1}

Output: 18

Explanation: The thief should steal from houses 10 + 8

**Recursion :**  
**TimeComplexity : O(2^n)**  
**Space Complexity = O(n)**





Actual Principle is =>

$$dp[i] = \max(profits[i] + dp[i-2], dp[i-1])$$

We have a constraint here array size is always less than the dp size. Leads to index OutOf Bounds Exception.

So track this way

Current 'i' element profit would be stored in  $dp[i+1]$ .

So principle is

$$dp[i+1] = \text{Math.max}(profits[i]+dp[i-1], dp[i]);$$

max(includeProfit,  
excludeProfit)

$$\max(5+0,1) = 5$$

max(includeProfit,  
excludeProfit)

$$\max(3+1,5) = 5$$

```
dp[0] = 0;
dp[1] = profits[0];
// When there is single House , we just take profit[0].
```

```
// Current 'i' element profit would be stored in dp[i+1]
for(int i = 1 ; i < profits.length;i++)
{
    dp[i+1] = Math.max(profits[i]+dp[i-1],dp[i]);
}
```

## Back Tracking ::

int[] profits = {1,5,7,8,11,10,3,22};

We know that profit[i ], could be represented by  
dp[i +1].

So that in BackTracking we can say  
dp[i ] can be represented by profits[i-1] .

When  $dp[i] \neq dp[i-1]$ , it means you included  
current element i.e profits[i-1].

So update the totalProfit , move the index to i-2.  
Because by including current element you got  
the MaxProfit, look back to other possibilities in  
same direction.

When  $dp[i] == dp[i-1]$ , it means you did not  
include current element so just move to previous  
row. " i - -".

int[] profits = {1,5,7,8,11,10,3,22};

$dp[i+1] = \text{Math.max}(\text{profits}[i]+dp[i-1], dp[i]);$

Index	Value	Step
0	0 = dp[0]	Step0
1	1 = dp[1]= 1	Step1
2	5 = dp[2]= 5	Step2
3	7 = dp[3]= 8	Step3
4	8 = dp[4]= 13	Step4
5	11 = dp[5]= 19	
6	10 = dp[6]= 23	
7	3 = dp[7]= 23	
8	0 = dp[8]= 23	

MaxProfit = 45

Selected Profits :{ 22 , 10 , 8 , 5}

We have Items, every Item has weight and profit.

Unbounded Knapsack Pattern

Fixed Capacit

### Maximum Ribbon Cut:

We are given a ribbon of length 'n' and a set of possible ribbon lengths a,b or c. We need to cut the ribbon into the maximum number of pieces that comply with the above-mentioned possible lengths. Write a method that will return the count of pieces.

n: 5

Ribbon Lengths: a=2, b=3, c=5

Output: 2

Explanation: Ribbon pieces will be {2,3}.

n: 7

Ribbon Lengths: a=2, b=3

Output: 3

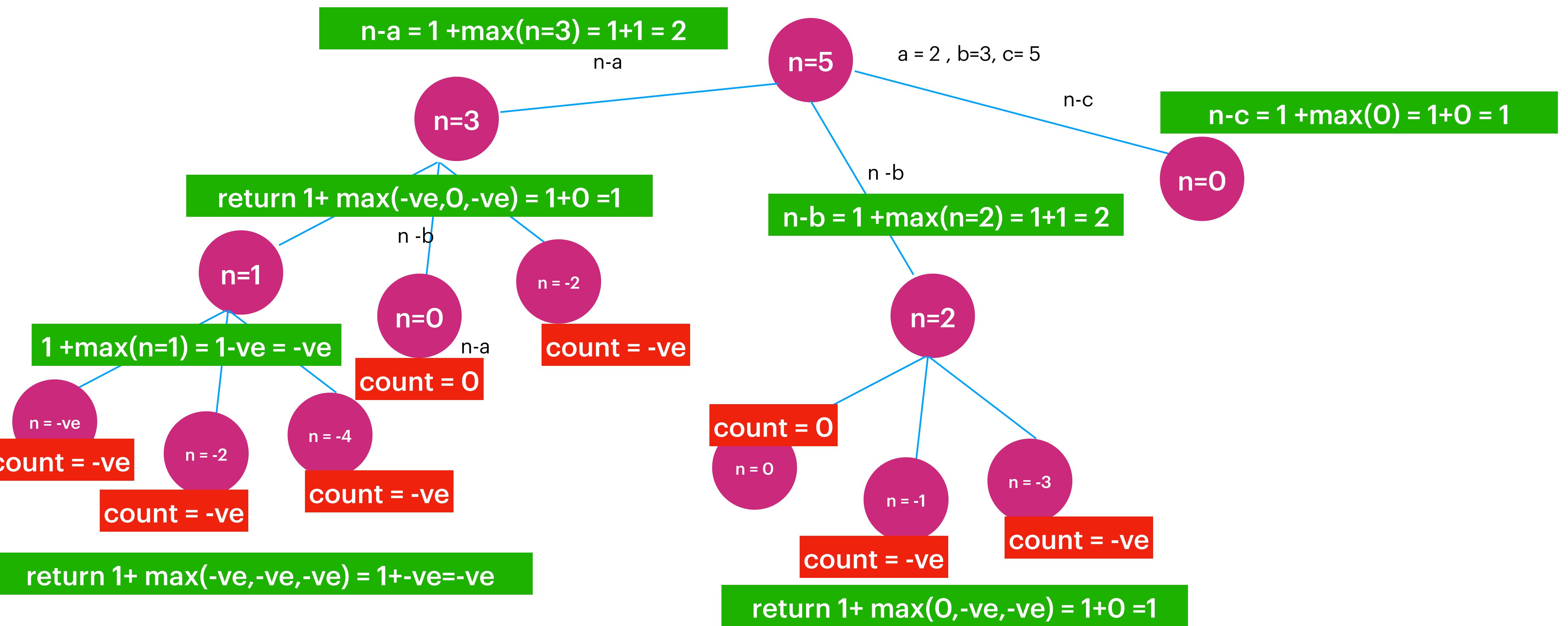
Explanation: Ribbon pieces will be {2,2,3}.

n: 13

Ribbon Lengths: a=3, b=5, c=7

Output: 3

Explanation: Ribbon pieces will be {3,3,7}.



Finally maximum of  $(n-a, n-b, n-c) = (2, 1, 1) = 2$

## Fibonacci Pattern



Express 'n' as the sum of 1, 3, or 4.

Given a number 'n', implement a method to count how many possible ways there are to express 'n' as the sum of 1, 3, or 4.

n : 4

Number of ways = 4

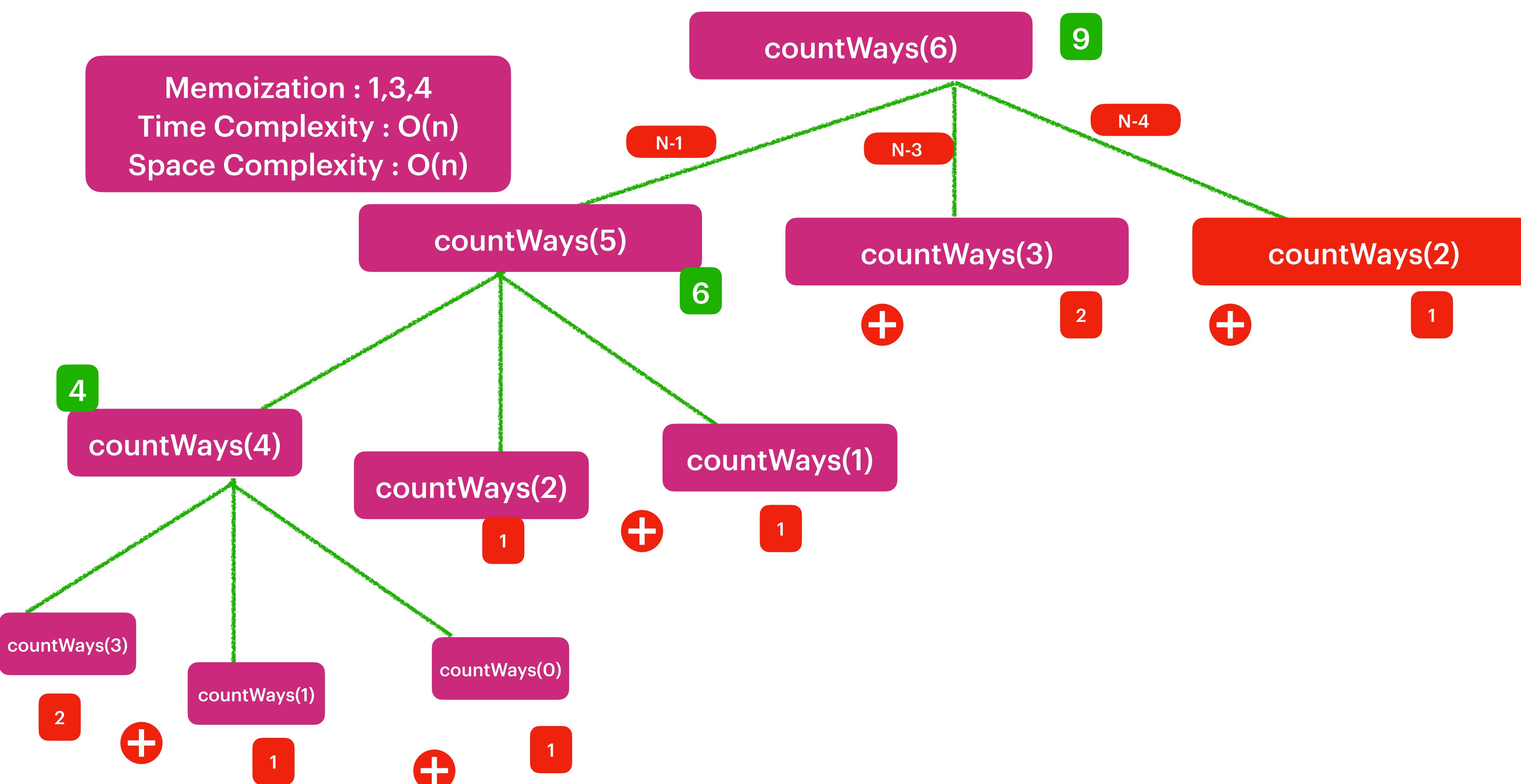
Explanation: Following are the four ways we can express 'n' : {1,1,1,1}, {1,3}, {3,1}, {4}

n : 5

Number of ways = 6

Explanation: Following are the six ways we can express 'n' : {1,1,1,1,1}, {1,1,3}, {1,3,1}, {3,1,1}, {1,4}, {4,1}

Memoization : 1,3,4  
Time Complexity : O(n)  
Space Complexity : O(n)



**Tabulation**

**Time Complexity : O(n)**

**Space Complexity : O(n)**

$dp[i] = dp[i-1] + dp[i-3] + dp[i-4];$

**What => DP is an Algorithm. Its Solution for subProblem overlapping.**

**Why is this been introduced => To reduced solving problem repeatedly so obviously it decreases the TimeComplexity.**

**When => Choice + Optimisation**

**How to implement => Can be applied any Data Structure.**

For Data Access Array is recommend ::

Time Complexity :

If we know the index number then its = O(1)

Otherwise Iteration is needed so O(n)

Space Complexity : O(n)

Data Deletion =>

Time Complexity :

when you delete an element  
all the elements should be  
moved an index before.

It's a costlier operation.

Time Complexity : O(n)

For Data Deletion Array is  
not recommended.

4000addr = 10  
4004addr = 20  
4008addr = 30  
4012addr = 40

int[] a = {10,20,30,40};

$$a[0] = 4000 + 0*4 = 4000 \text{ addr} = 10$$

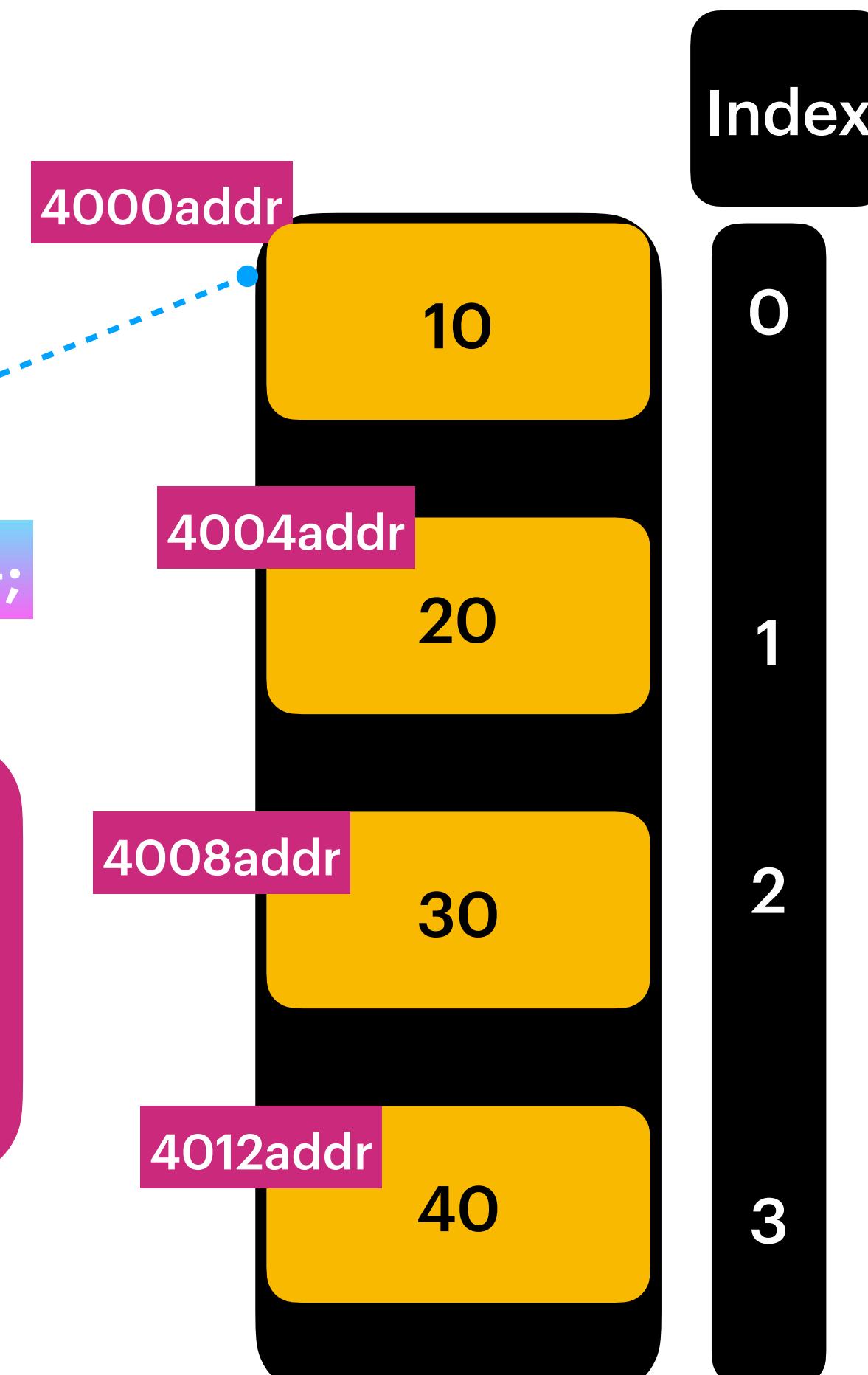
$$a[1] = 4000 + 1*4 = 4004 \text{ addr} = 20$$

$$a[2] = 4000 + 2*4 = 4008 \text{ addr} = 30$$

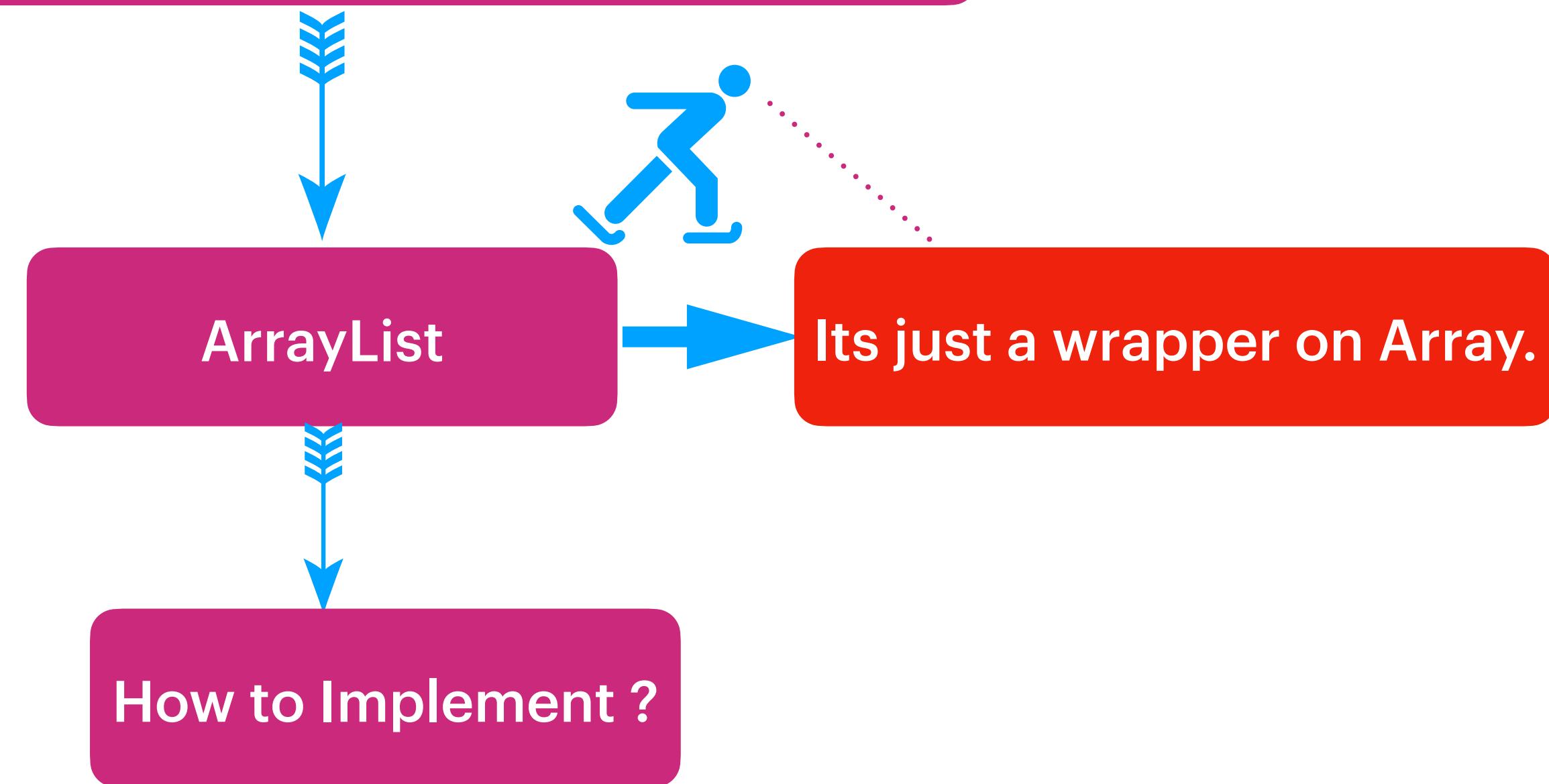
$$a[3] = 4000 + 3*4 = 4012 \text{ addr} = 40$$

It's On Array  
Array is fixed in size.

int[] a = new int[5];



## Lets Make size of array as Dynamic :::



	Time Complexity	Space Complexity
Add Element at last	O(1)	O(n)
Remove Element	O(n)	O(1)
Search Element	O(n)	O(1)
Get Element based On Index	O(1)	O(1)

## Remove Element In an Array

Given an integer array `nums` and an integer `val`, remove all occurrences of `val` in `nums` in-place. The relative order of the elements may be changed.

Input: `nums` = [3,2,2,3], `val` = 3  
Output: 2, `nums` = [2,2,\_,\_]

Explanation: Your function should return  $k = 2$ , with the first two elements of `nums` being 2. It does not matter what you leave beyond the returned  $k$  (hence they are underscores).

Input: `nums` = [0,1,2,2,3,0,4,2], `val` = 2  
Output: 5, `nums` = [0,1,4,0,3,\_,\_,\_]

Explanation: Your function should return  $k = 5$ , with the first five elements of `nums` containing 0, 0, 1, 3, and 4.

Note that the five elements can be returned in any order.  
It does not matter what you leave beyond the returned  $k$  (hence they are underscores).

## Remove Duplicates from Sorted Array

Given an array of integers arr, return true if and only if it is a valid mountain array.

Input: nums = [1,1,2]

Output: 2, nums = [1,2,\_]

Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

Input: nums = [0,0,1,1,1,2,2,3,3,4]

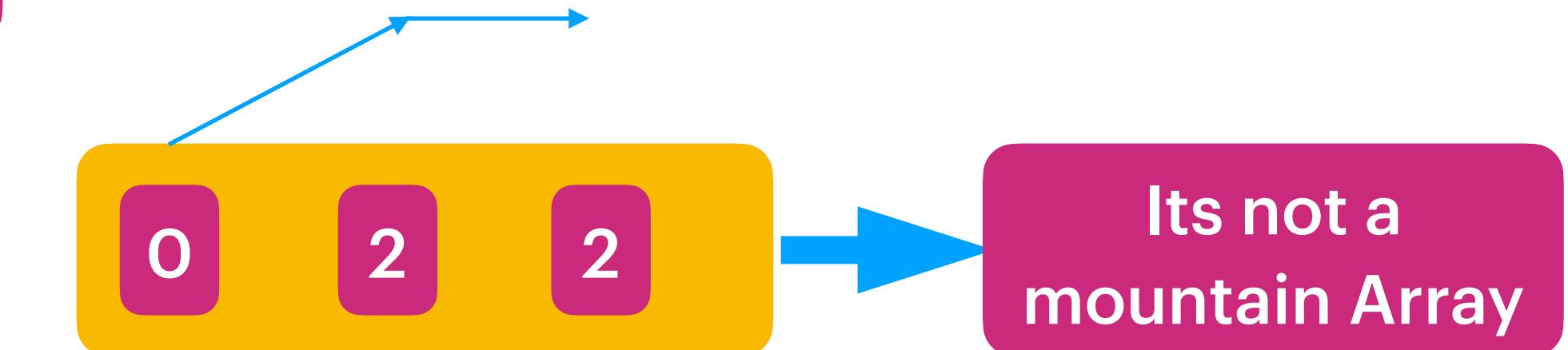
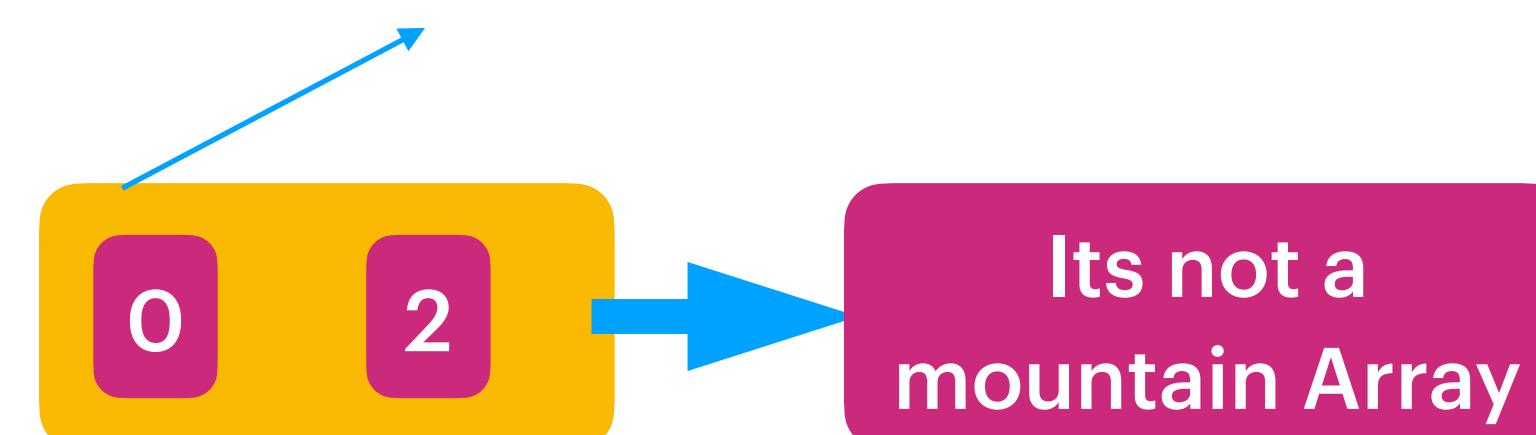
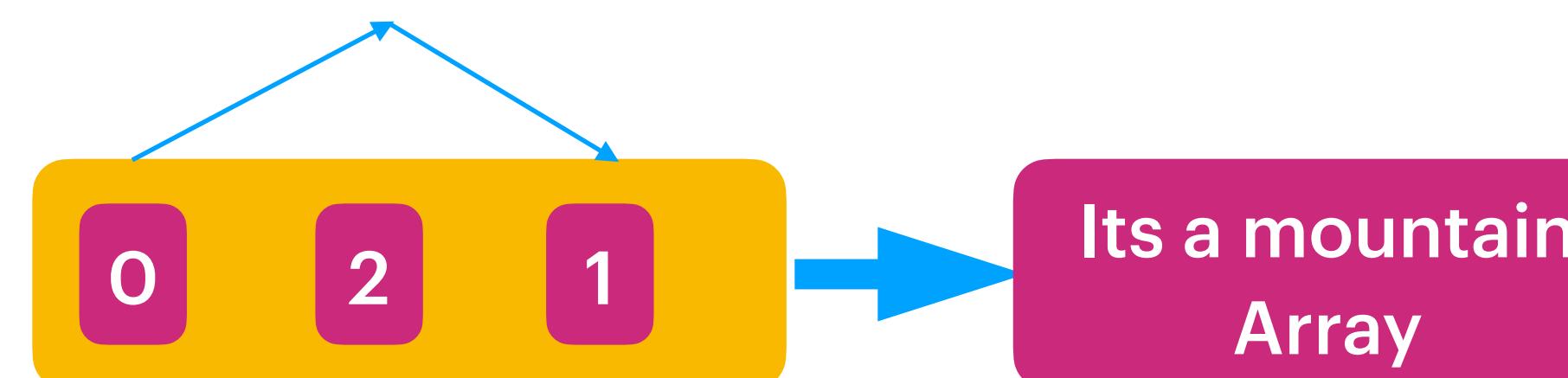
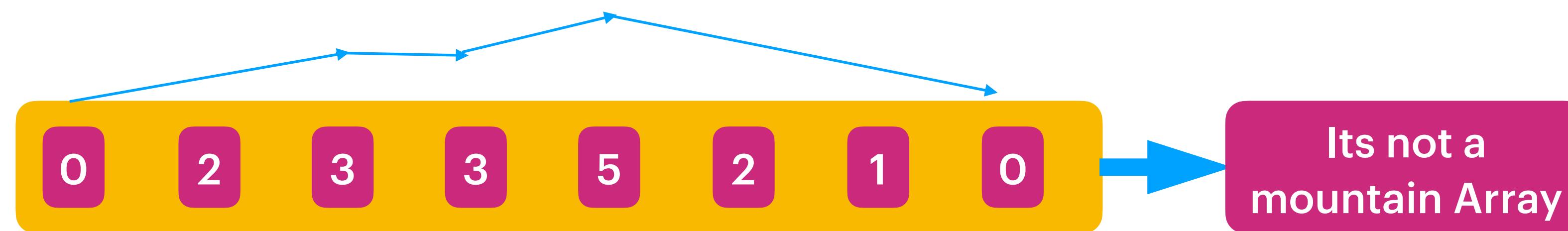
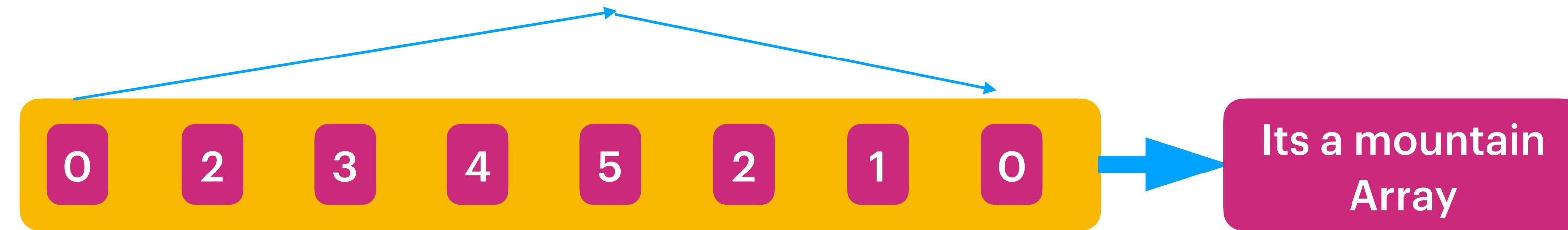
Output: 5, nums = [0,1,2,3,4,\_,\_,\_,\_,\_]

Explanation: Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

## Valid Mountain Array

Given an array of integers arr, return true if and only if it is a valid mountain array.



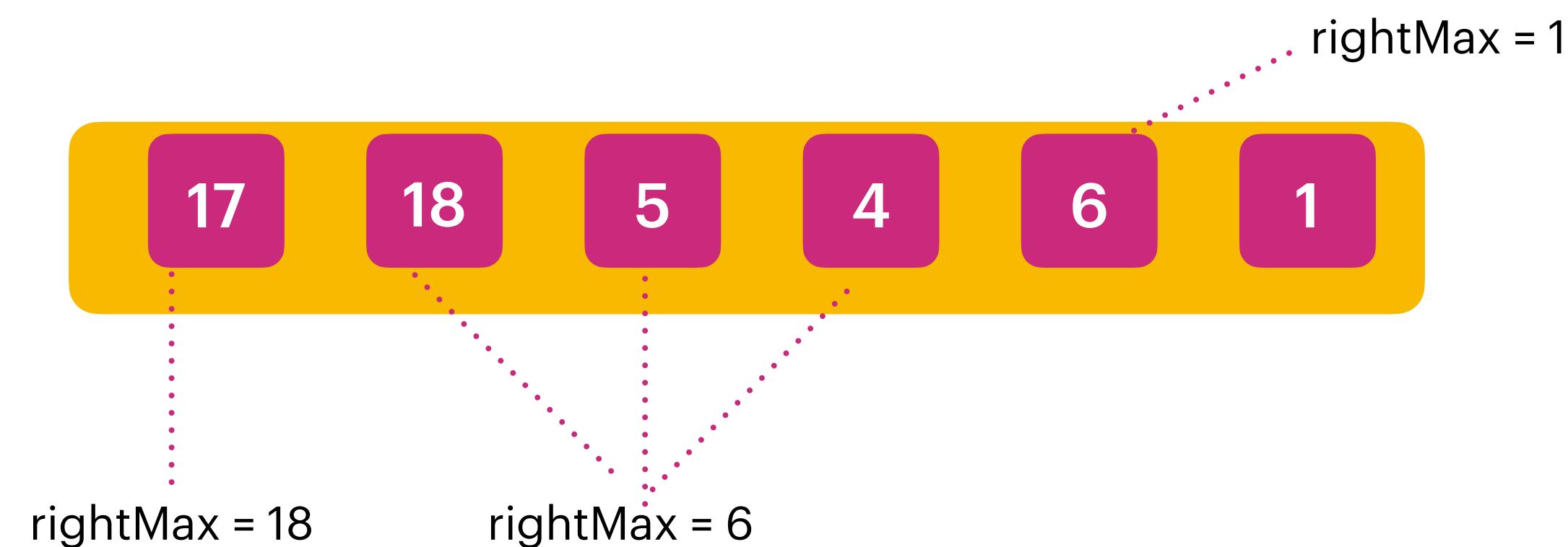
## Replace Elements with Greatest Element on Right Side !!!

Given an array arr, replace every element in that array with the greatest element among the elements to its right, and replace the last element with -1.

After doing so, return the array.

Input: arr = [17,18,5,4,6,1]

Output: [18,6,6,6,1,-1]



## Move Zeroes !!!

Given an integer array `nums`, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this in-place without making a copy of the array.

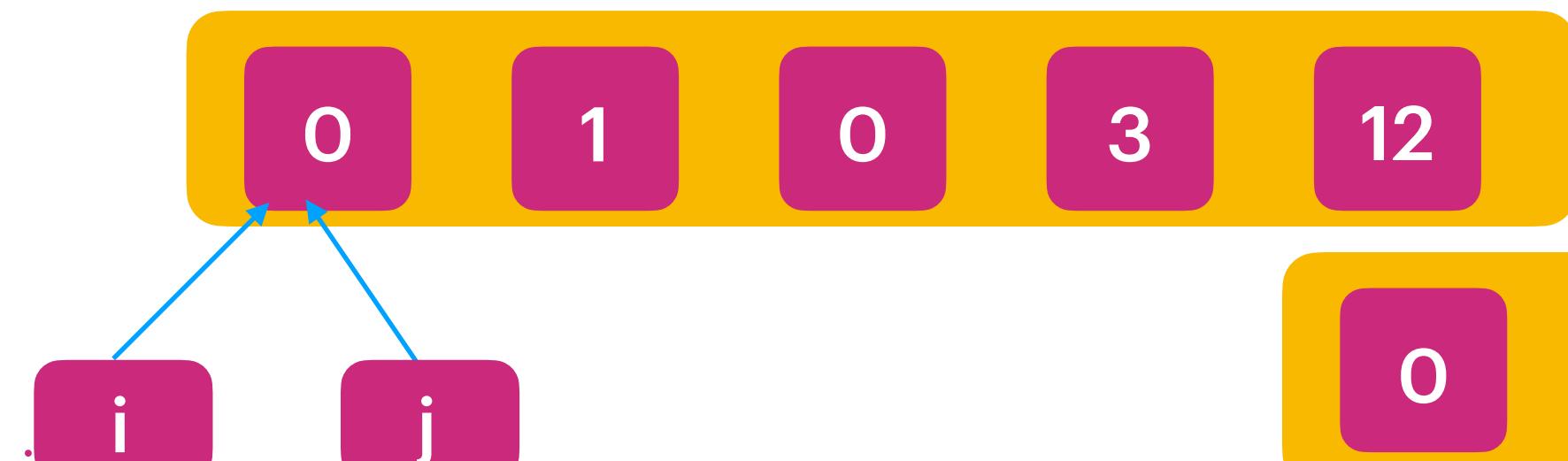
Input: `nums = [0,1,0,3,12]`

Output: `[1,3,12,0,0]`

Input: `nums = [1,2,3]`

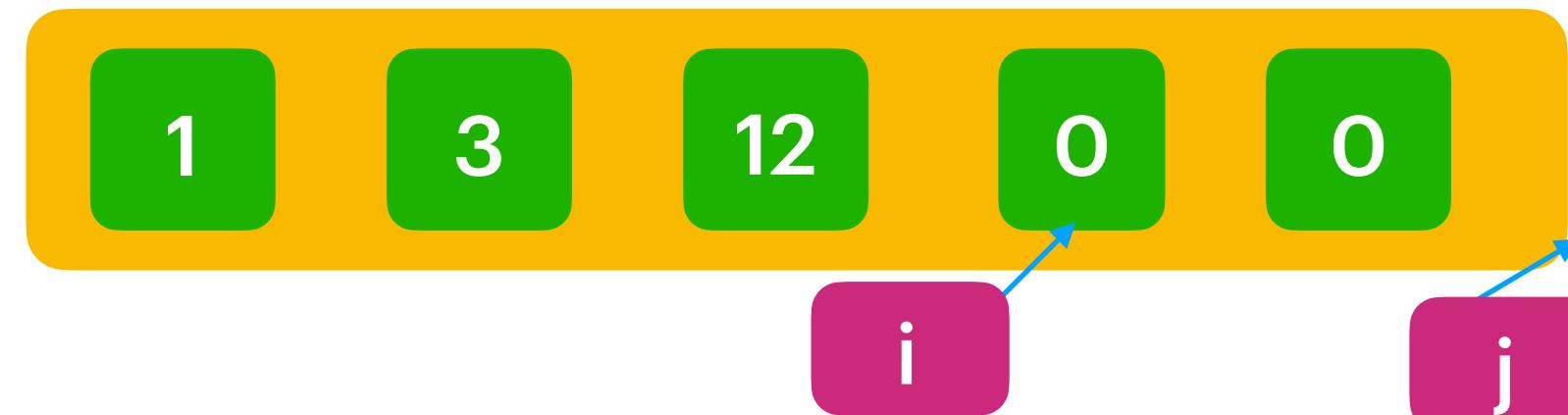
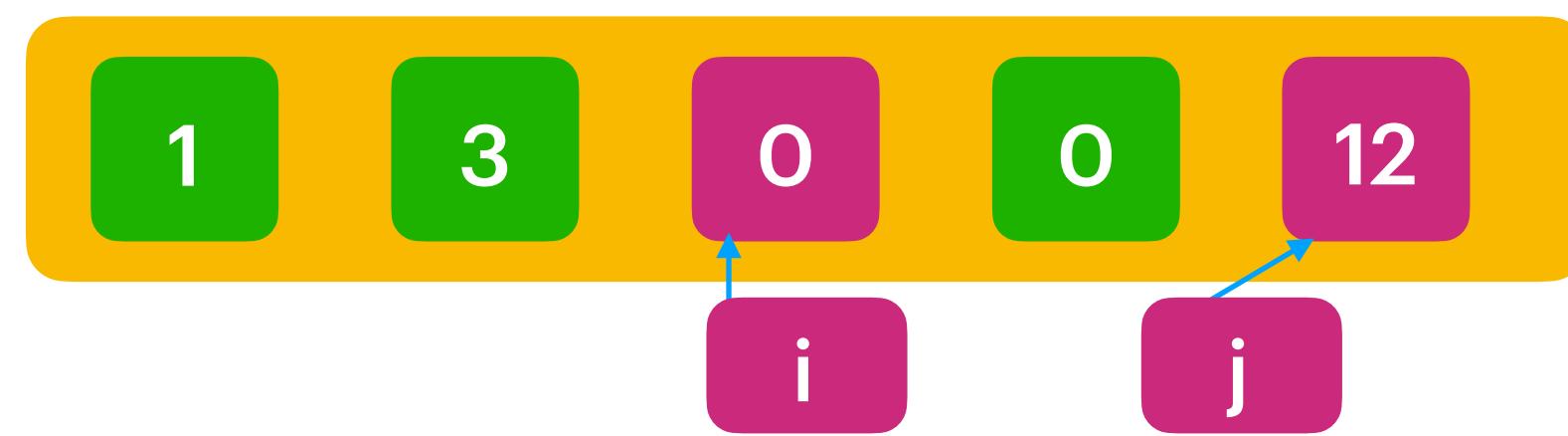
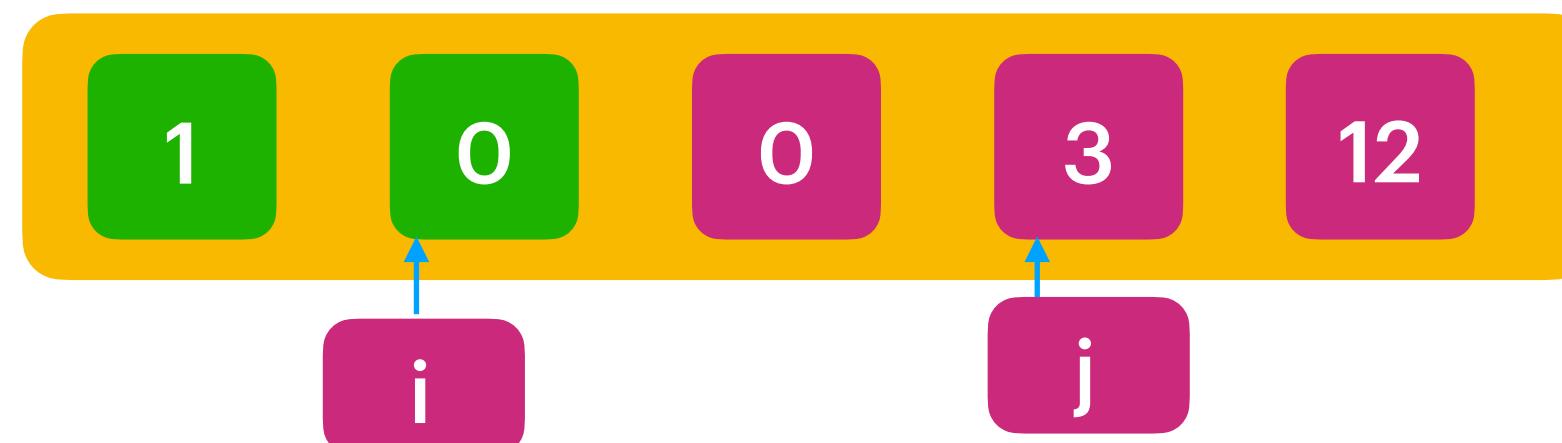
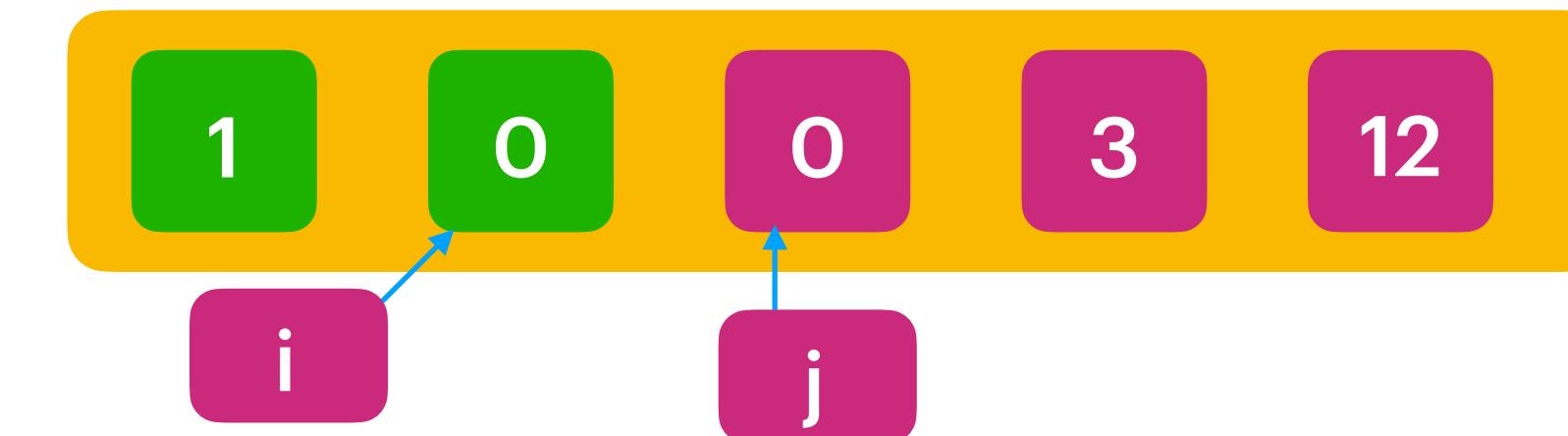
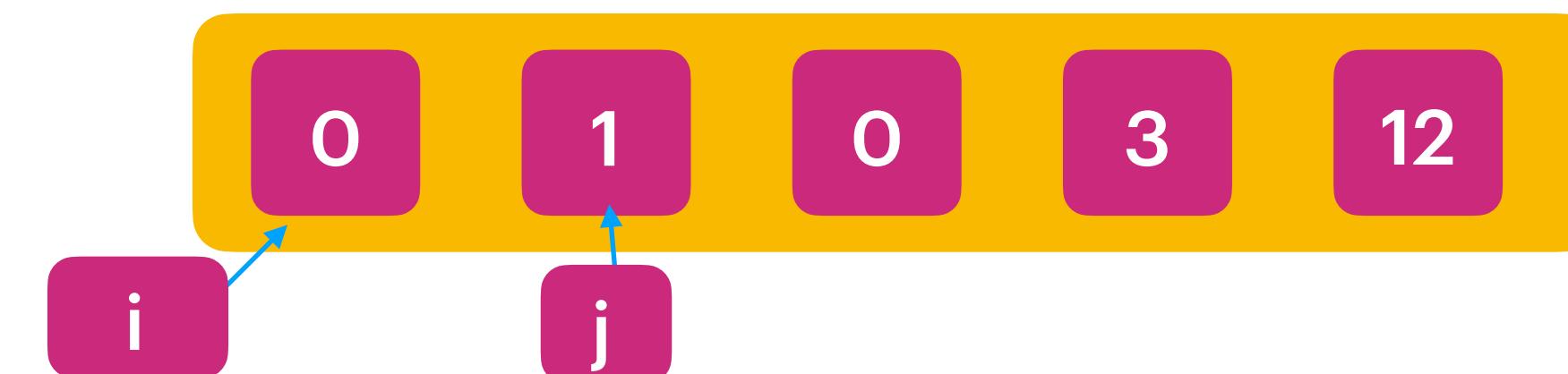
Output: `[1,2,3]`

when  $\text{arr}[j] \neq 0$  then  
swap i,j.



Move i to right only when there is a swap

Move j to right till n-1 also do  
swap when  $\text{arr}[j] \neq 0$ .



## Find All Numbers Disappeared in an Array:

Given an array `nums` of  $n$  integers where  $\text{nums}[i]$  is in the range  $[1, n]$ , return an array of all the integers in the range  $[1, n]$  that do not appear in `nums`.

Input: `nums` = [4,3,2,7,8,2,3,1]

Output: [5,6]

Input: `nums` = [1,1]

Output: [2]



When the numbers are 1 to  $n$  then always we can represent value  $n$  with  $n-1$  index in an array.

At index 3 & 4 elements are positive so {4,5} are missing !!!



## Count even Digits in a Array ( Apply Math)

Given an array `nums` of integers, return how many of them contain an even number of digits.

**Input:** `nums = [12,345,2,6,7896]`

**Output:** 2

**Explanation:**

12 contains 2 digits (even number of digits).

345 contains 3 digits (odd number of digits).

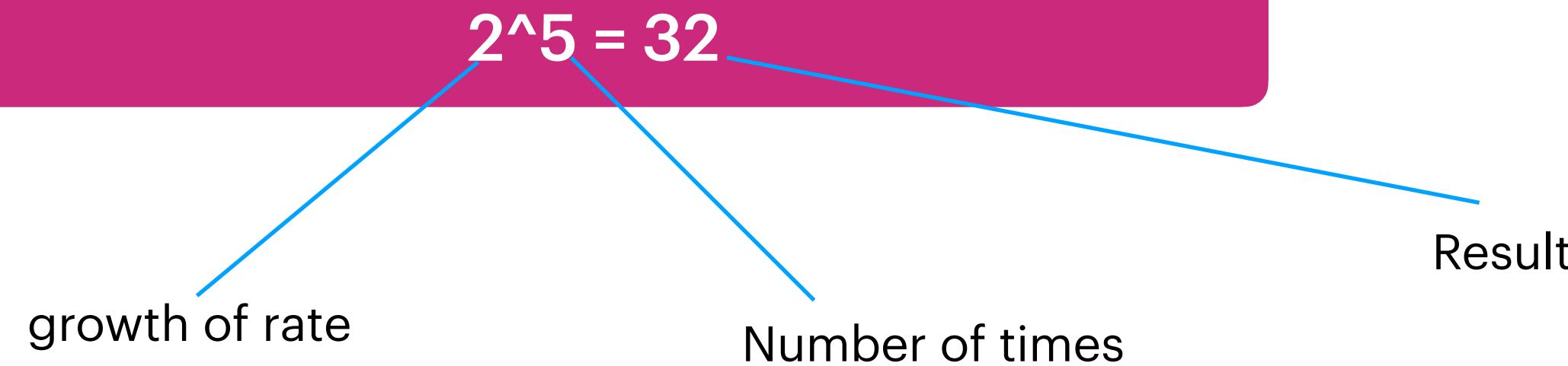
2 contains 1 digit (odd number of digits).

6 contains 1 digit (odd number of digits).

7896 contains 4 digits (even number of digits).

Therefore only 12 and 7896 contain an even number of digits.

## Lets understand the pattern logarithms



$$\log_2(16) = 4$$

$$\log_2(34) = 5.000123$$

$$\log_{10}(100) = 2$$

$$\log_{10}(999) = 2.999$$

$$\log_{10}(1000) = 3$$

$$\log_{10}(9999) = 3.9999$$

/\*

When digit is odd then logValue with base10  
would be even Integer,  
(Don't consider decimal)

$\log_{10}(100) \dots \log_{10}(999) = 2 \text{ to } 2.99999$   
\*/

/\*

When digit is even then logValue with base10  
would be odd Integer,  
(Don't consider decimal)

$\log_{10}(1000) \dots \log_{10}(9999) = 3 \text{ to } 3.99999$   
\*/

## Find Missing Number in given Array. (Apply Math)

Element would be in 1 to n in random order !!!

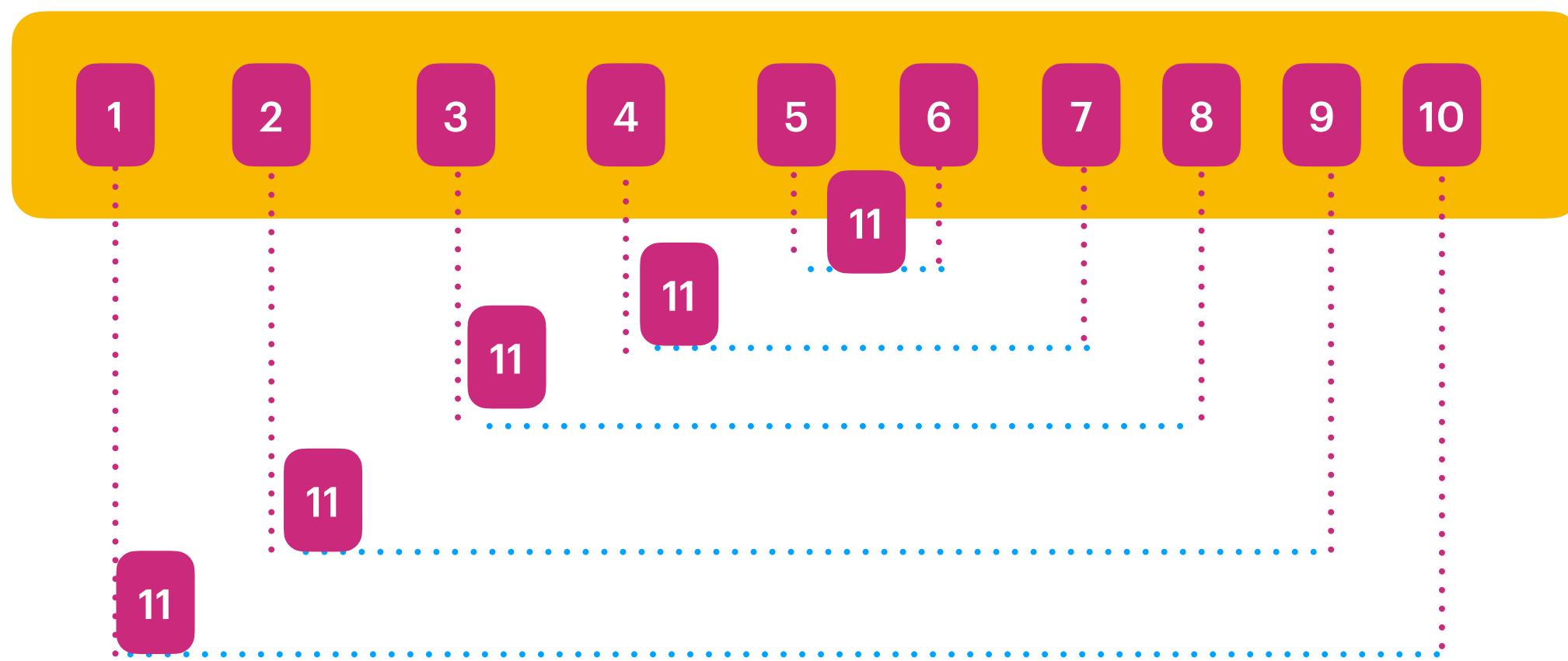
int[] arr = {1,3,4,5}, n=5

Missing element : 2

int[] arr = {1,2,4} , n=4

Missing element : 3

## Lets understand the pattern of natural numbers



## Missing Number - Hashing

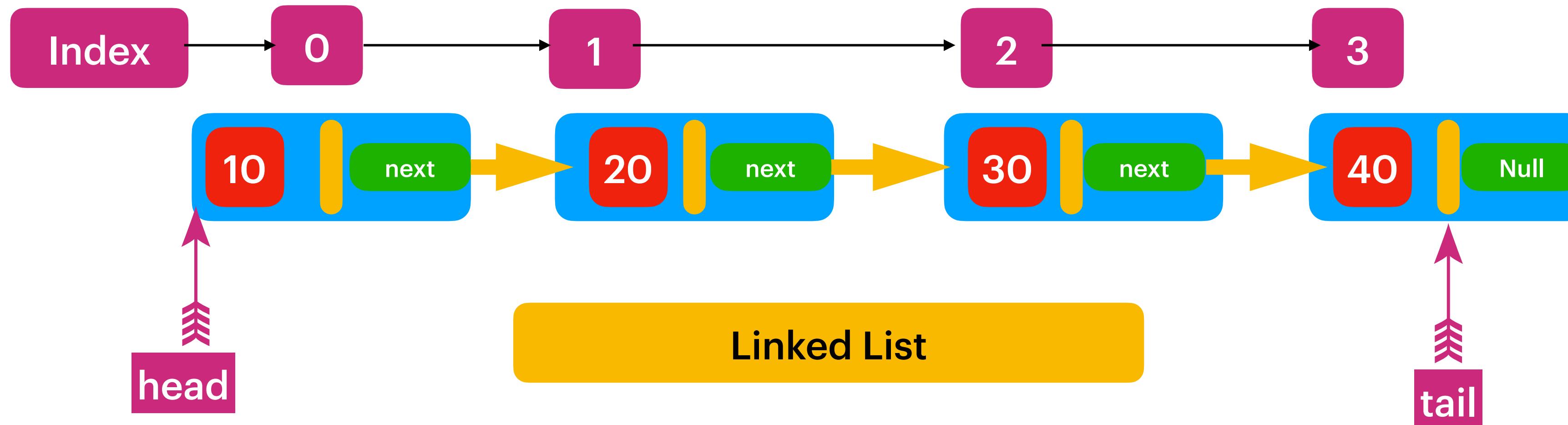
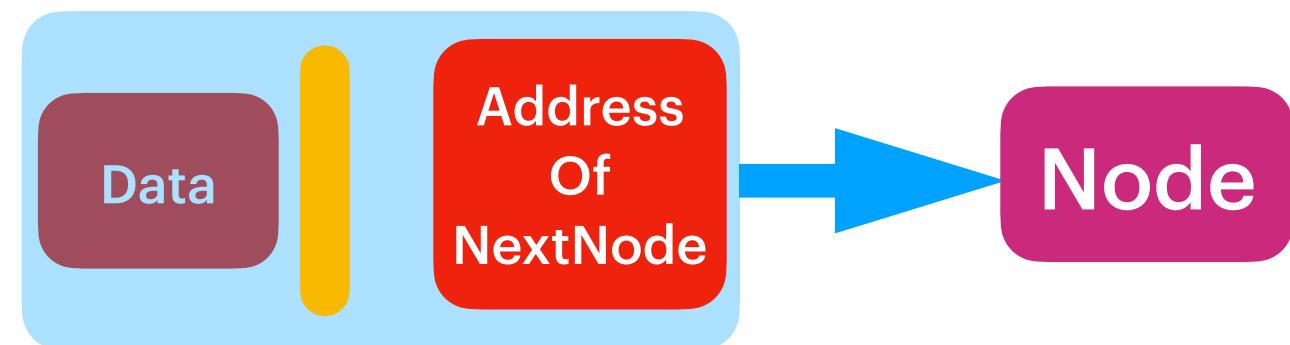
Given an array `nums` containing  $n$  distinct numbers in the range  $[0, n]$ , return the only number in the range that is missing from the array.

Input: `nums` = [3,0,1]  
Output: 2

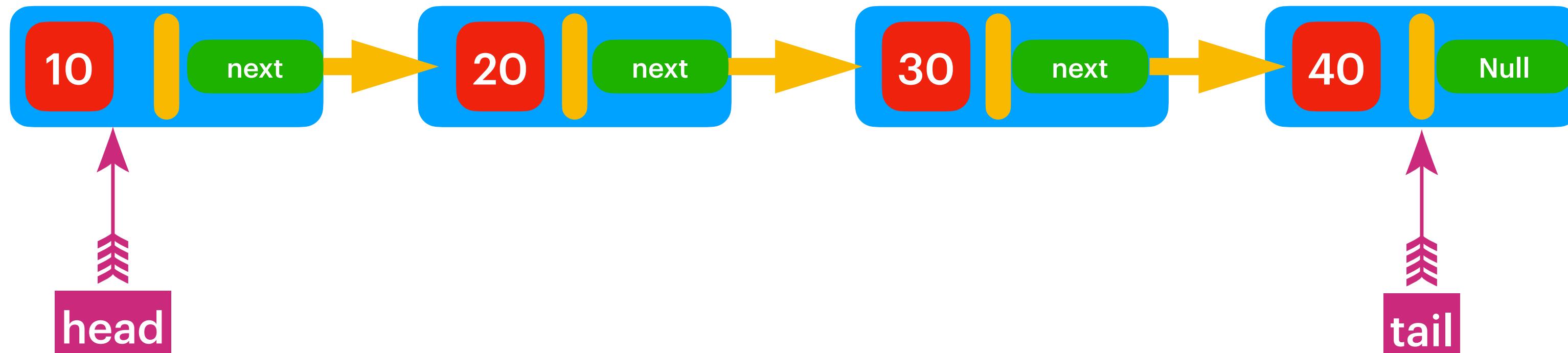
Explanation:  $n = 3$  since there are 3 numbers, so all numbers are in the range  $[0,3]$ . 2 is the missing number in the range since it does not appear in `nums`.

Input: `nums` = [9,6,4,2,3,5,7,0,1]  
Output: 8

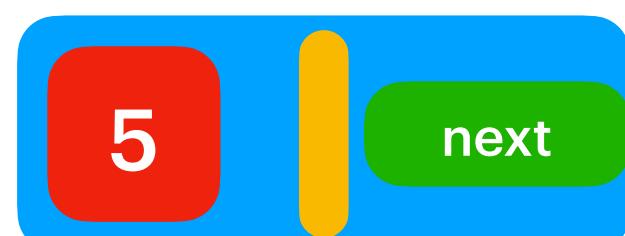
Explanation:  $n = 9$  since there are 9 numbers, so all numbers are in the range  $[0,9]$ . 8 is the missing number in the range since it does not appear in `nums`.



## Add to Head



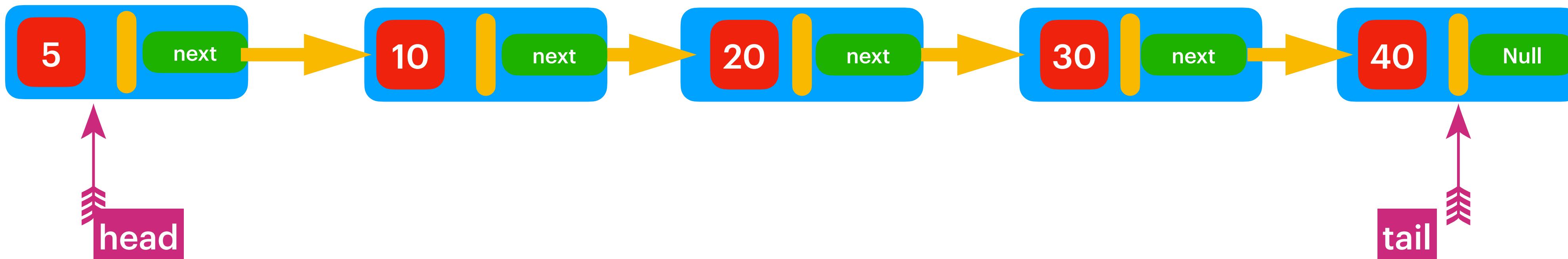
**newNode**



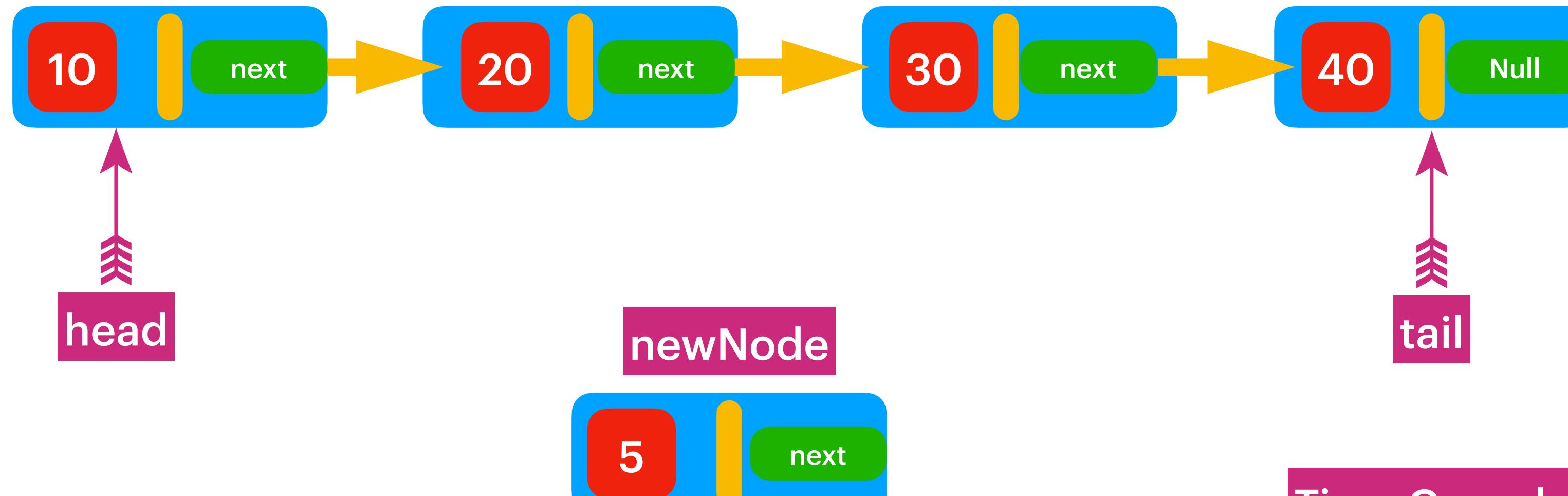
```
class Node {  
    int data,  
    Node next;  
}
```

Time Complexity = O(1)

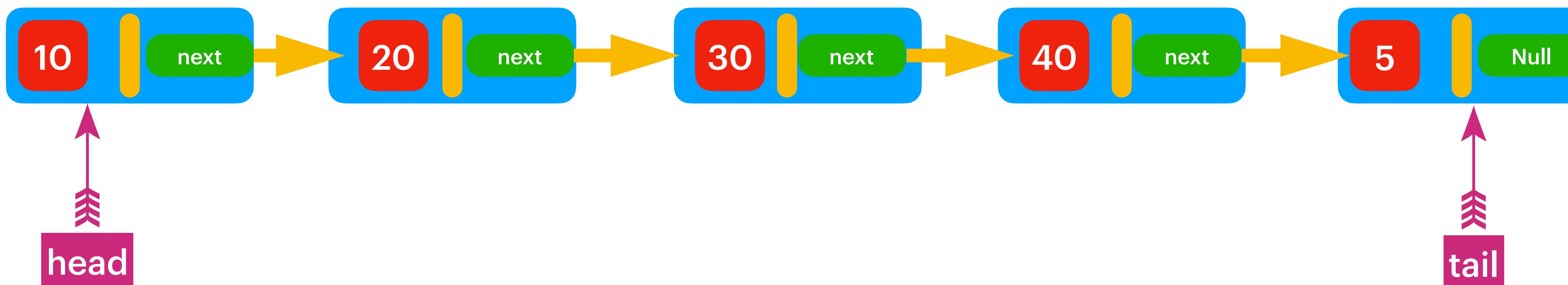
Link **newNode** next to **head** .  
Make **newNode** as **head**.



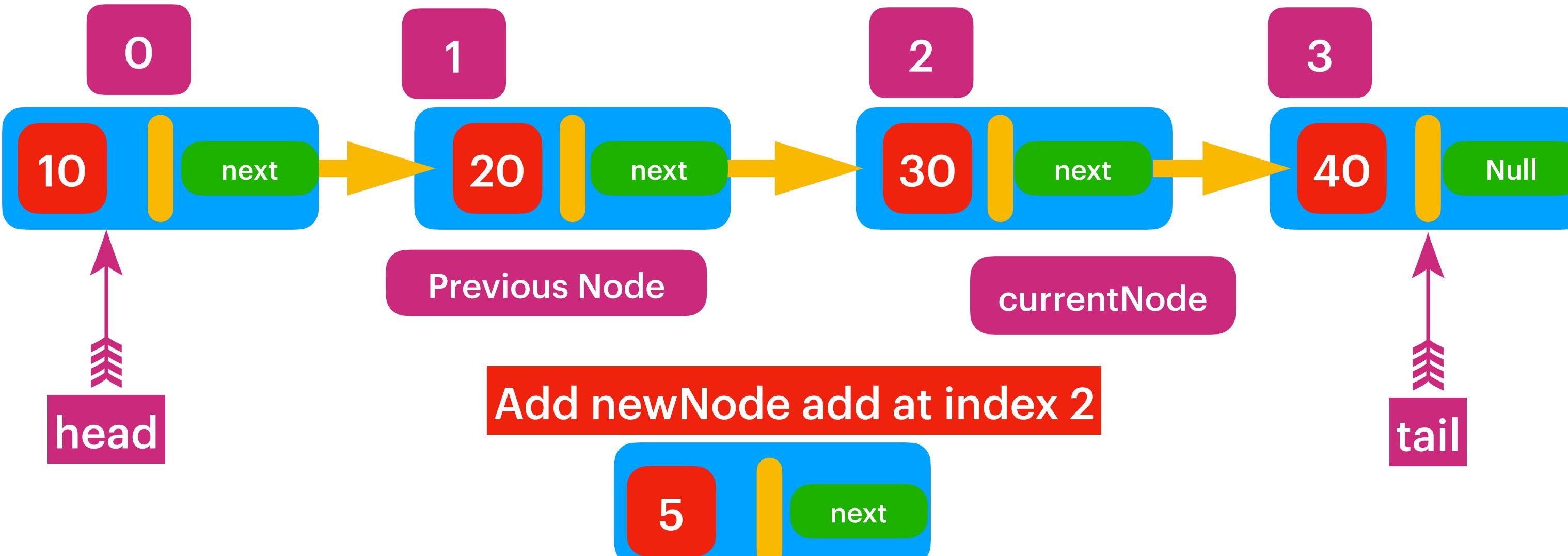
## Add Last



Time Complexity = O(1)

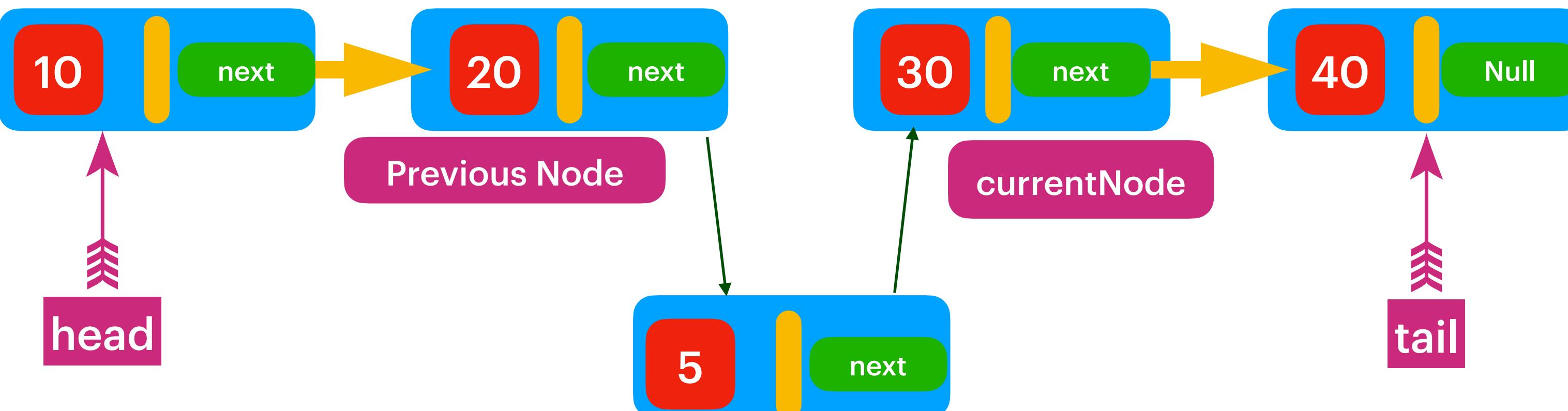


## Add in the Middle

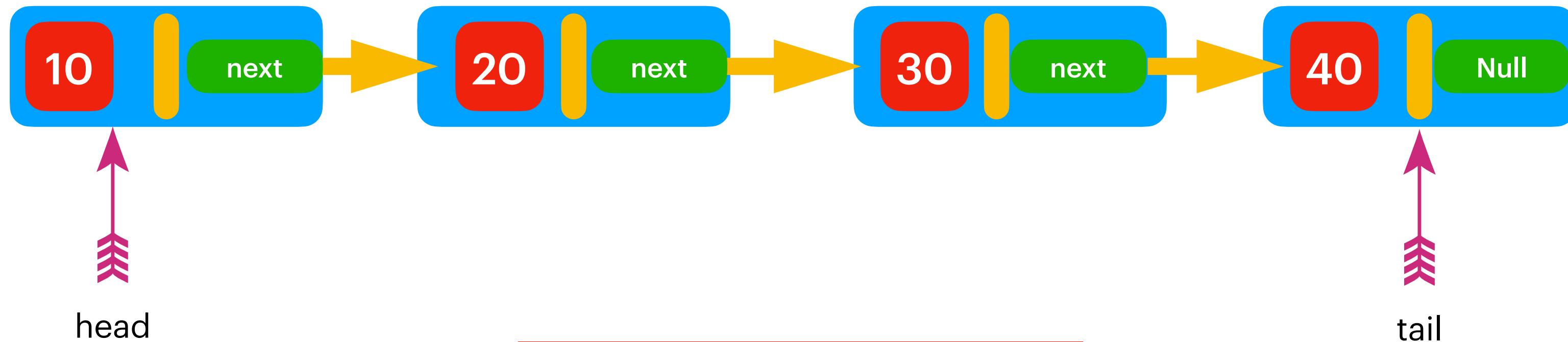


Link previousNode next to new Node.  
Link newNode next to currentNode.

Time Complexity : O(n)  
With respect to Shifting of element : O(1)

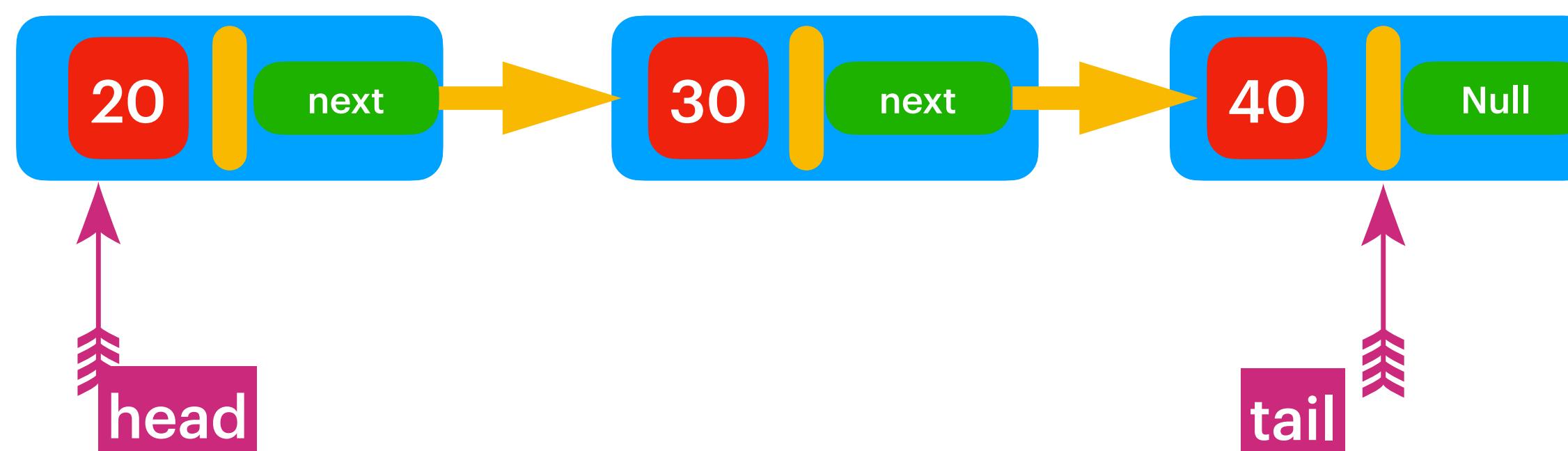


Delete Head

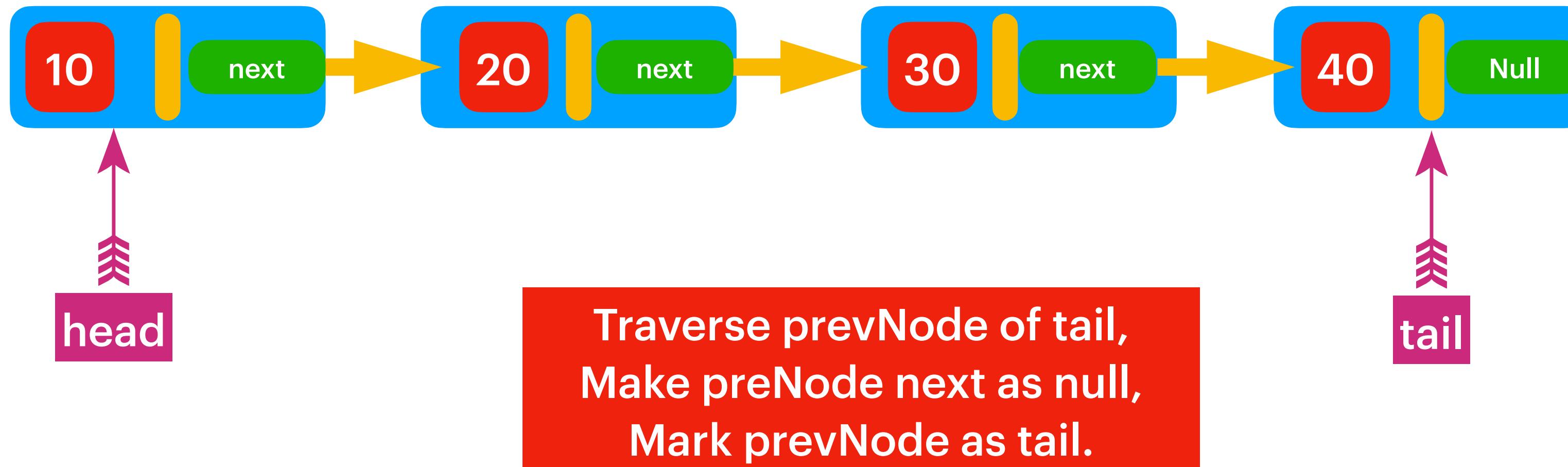


Link head.next to head

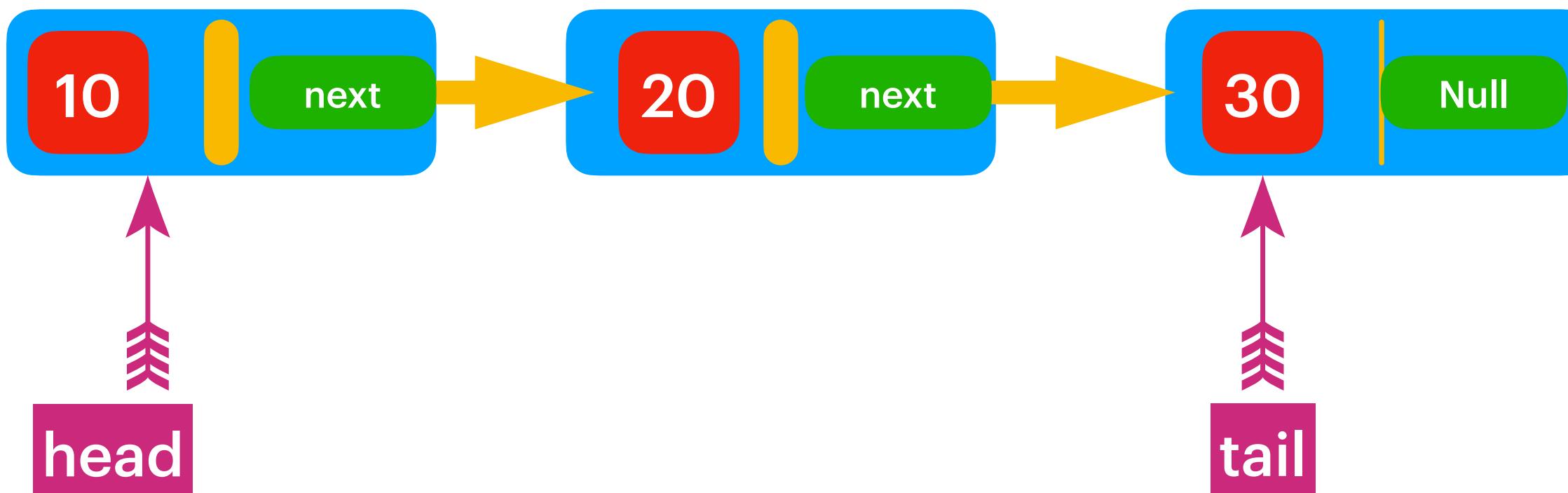
Time Complexity : O(1)



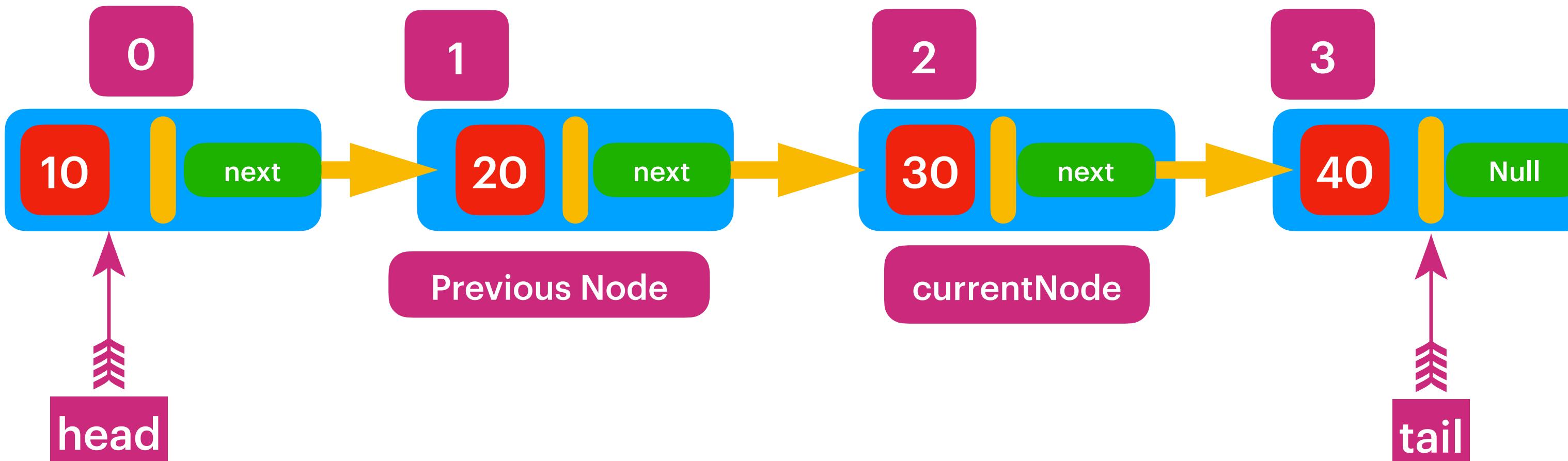
## Delete Tail



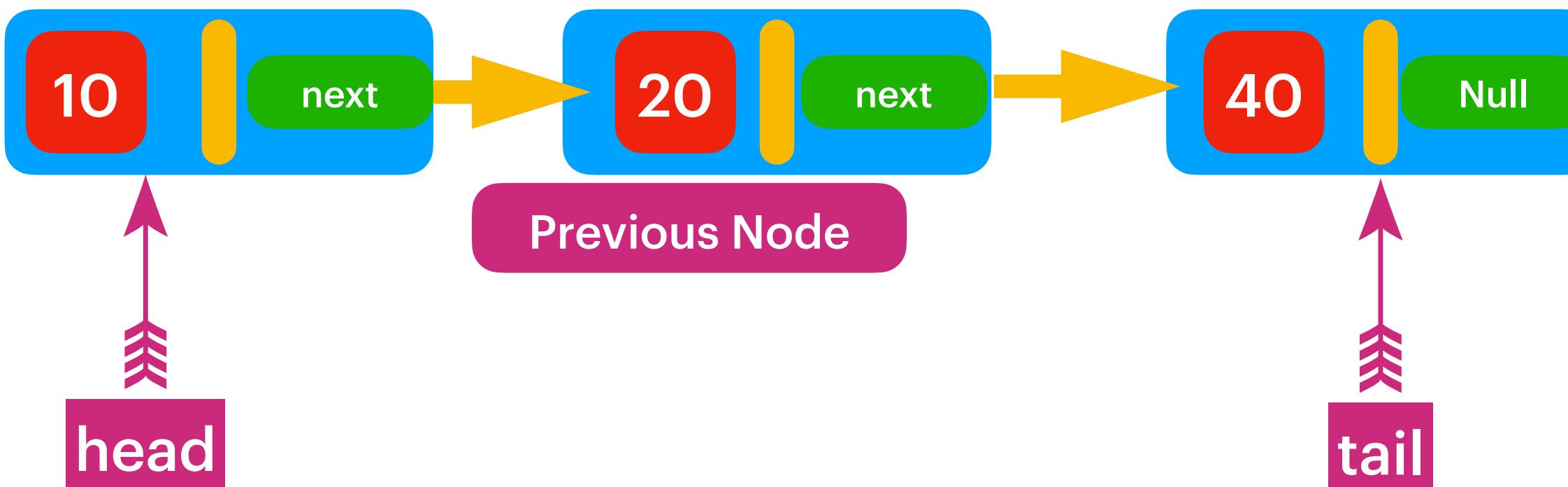
Time Complexity : O(n)



## Delete Middle



Link PreviousNode next to currentNode next.  
Mark currentNode next as null. So that current Node would be garbage collected.

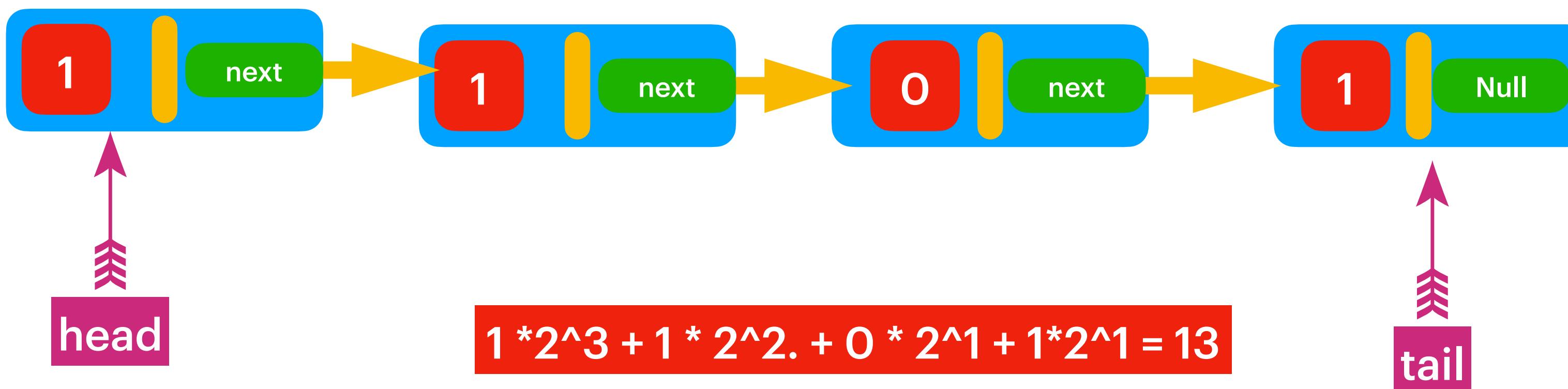


Time Complexity : O(n)  
With respect to Shifting of  
element : O(1)

## Convert Binary Number in a Linked List to Integer.

Given **head** which is a reference node to a singly-linked list. The value of each node in the linked list is either 0 or 1. The linked list holds the binary representation of a number.

Return the decimal value of the number in the linked list.



Input: head = [1,1,0,1]

Output: 13

Explanation: (1101) in base 2 = (3)

Input: head = [1,0,1]

Output: 5

Explanation: (101) in base 2 = (5)

Input: head = [1,0,0,0]

Output: 8

Explanation: (10000) in base 2 = (8)

Input: head = [0,0]

Output: 0

Explanation: (0000) in base 2 = 0

Input: head = [1]

Output: 1

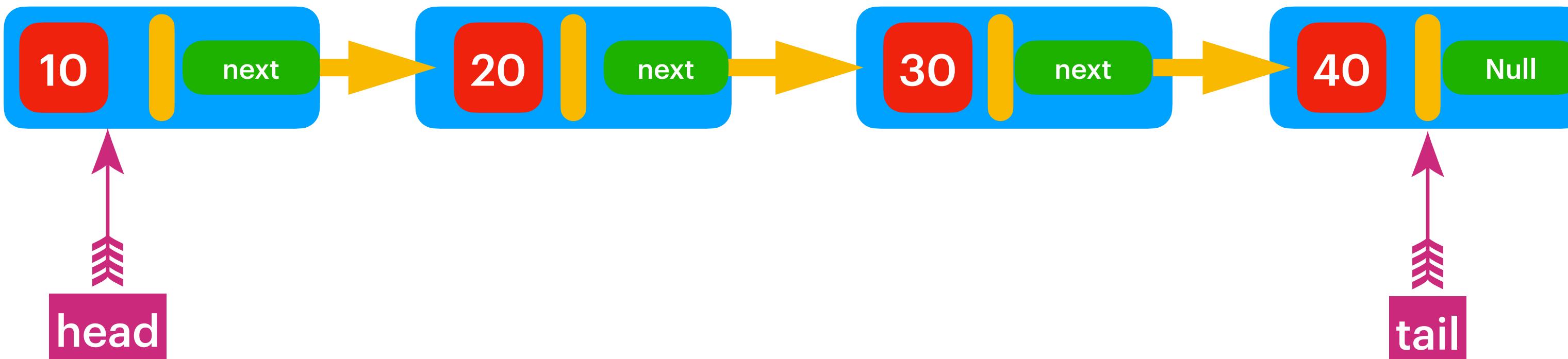
Explanation: (1) in base 2 = 1

## Print Immutable Linked List in Reverse

An immutable List can not be modified .

Ex: [10->20->30->40->null]

Output : 40,30,20,10



## Reverse The LinkedList

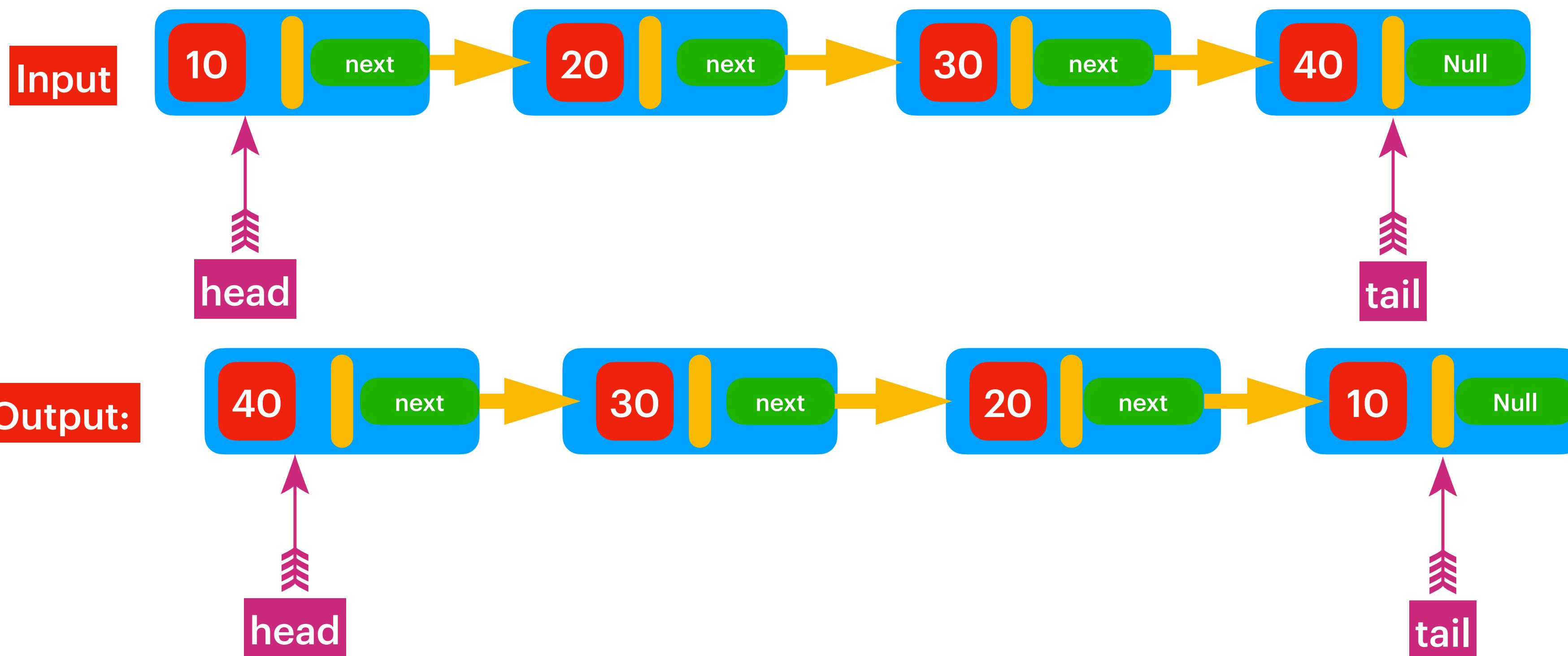


Hint :

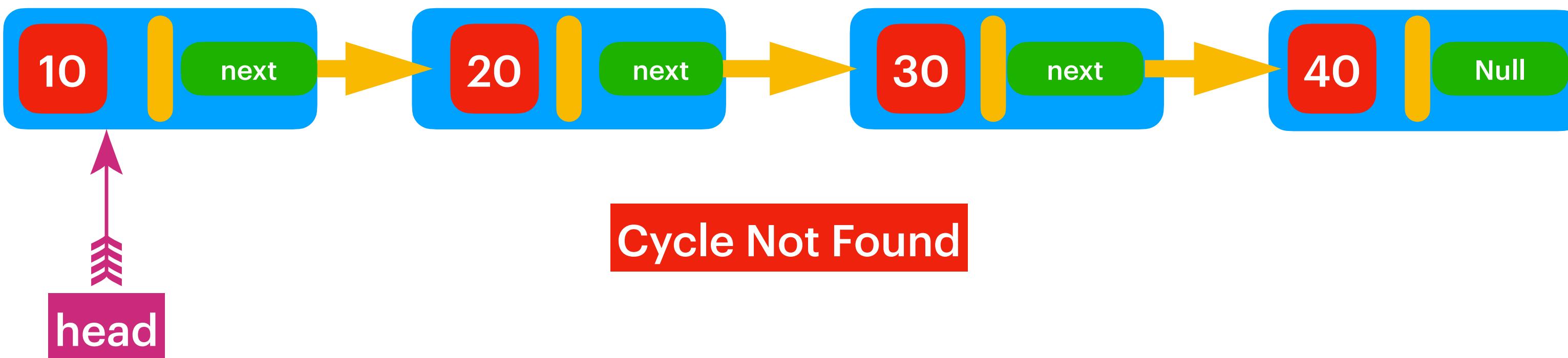
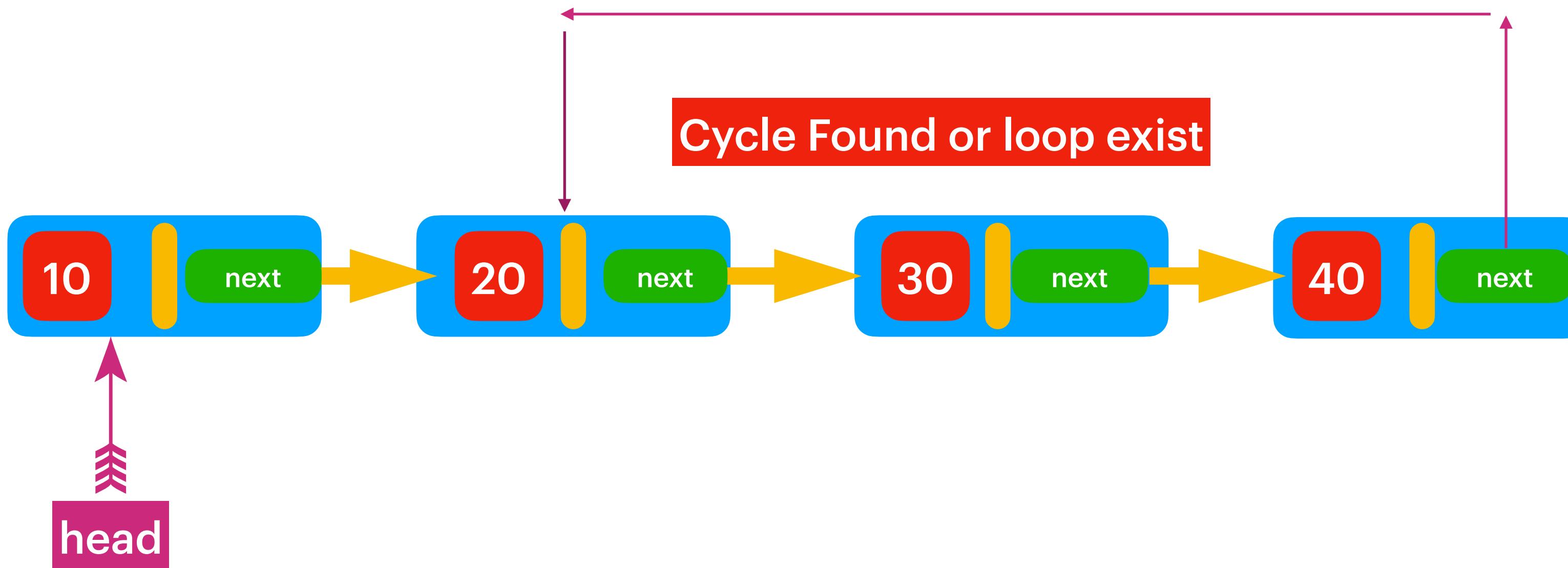
After reverse for the CurrentNode,  
previous Node is  
going to be the  
nextNode.

Ex Input: [10->20->30->40->null]

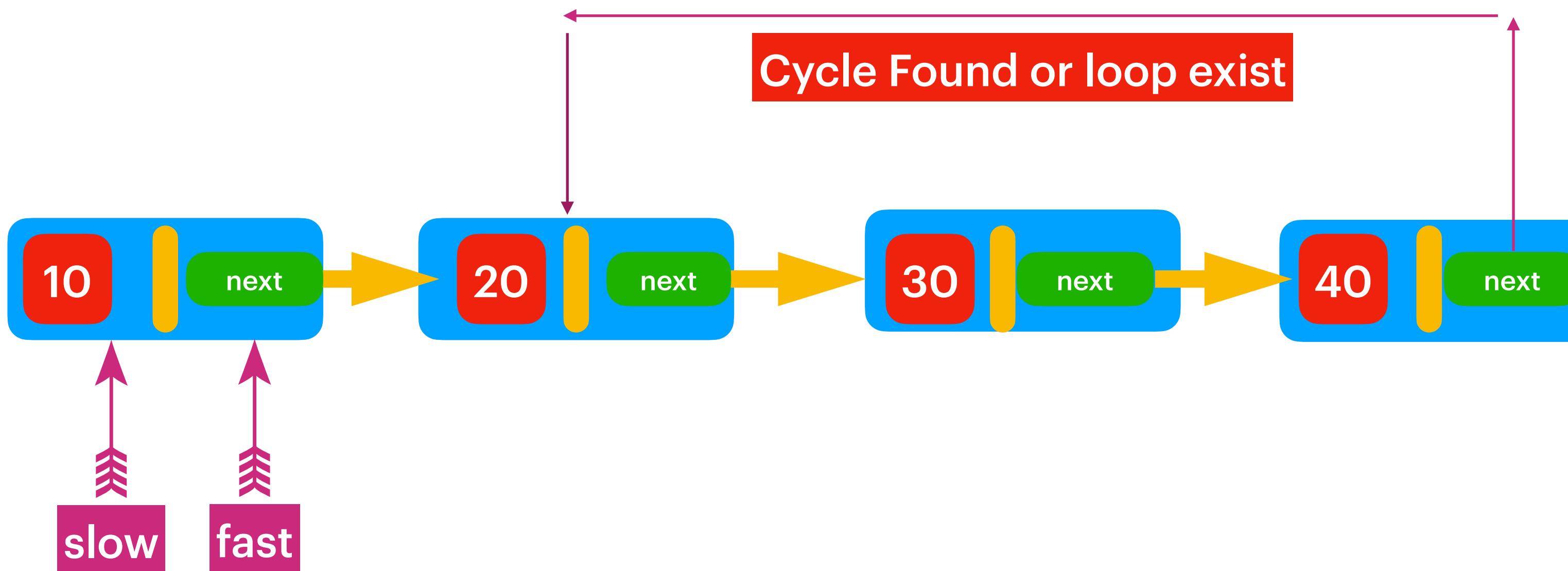
Output : [40->30->20->10->null]



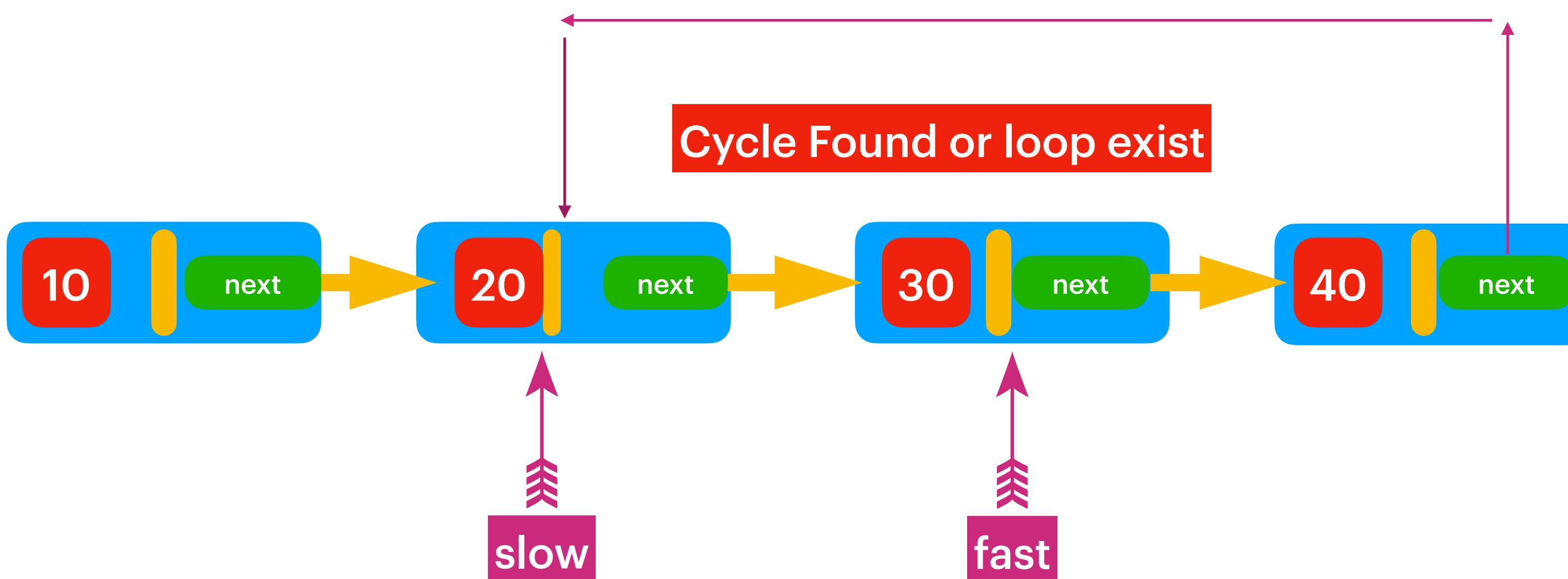
## Detect the Cycle In List



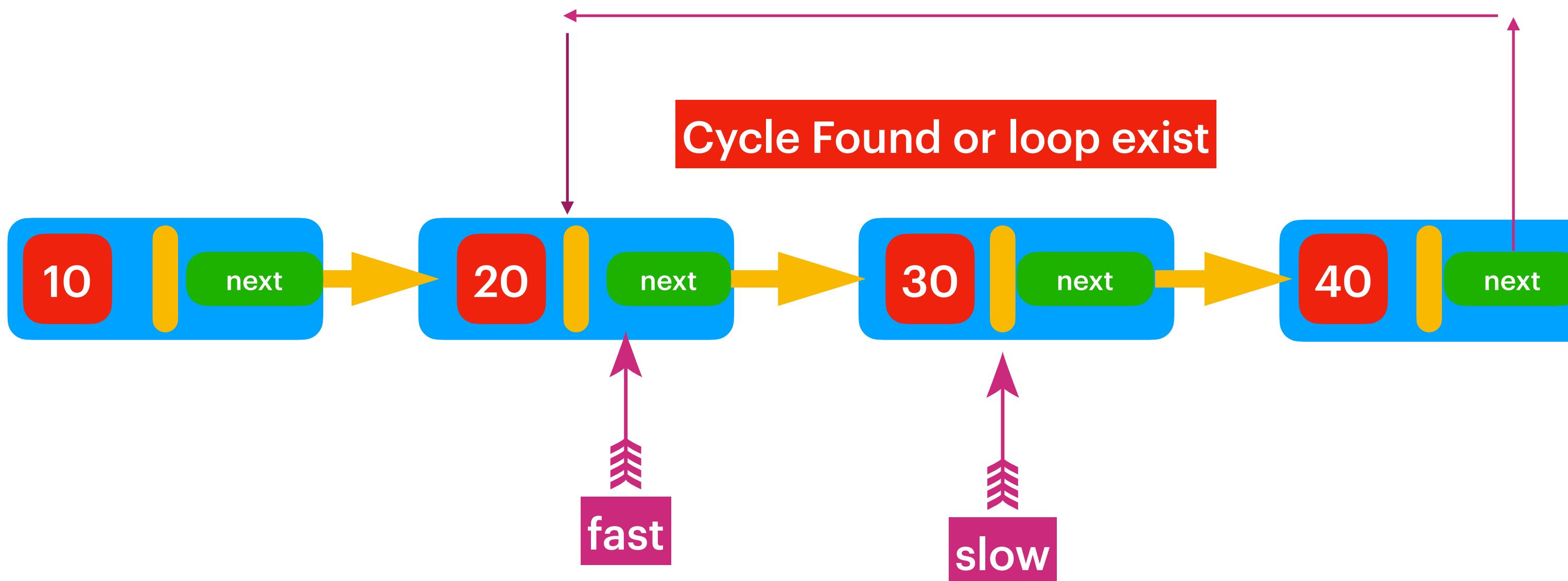
## Detect the Cycle In List



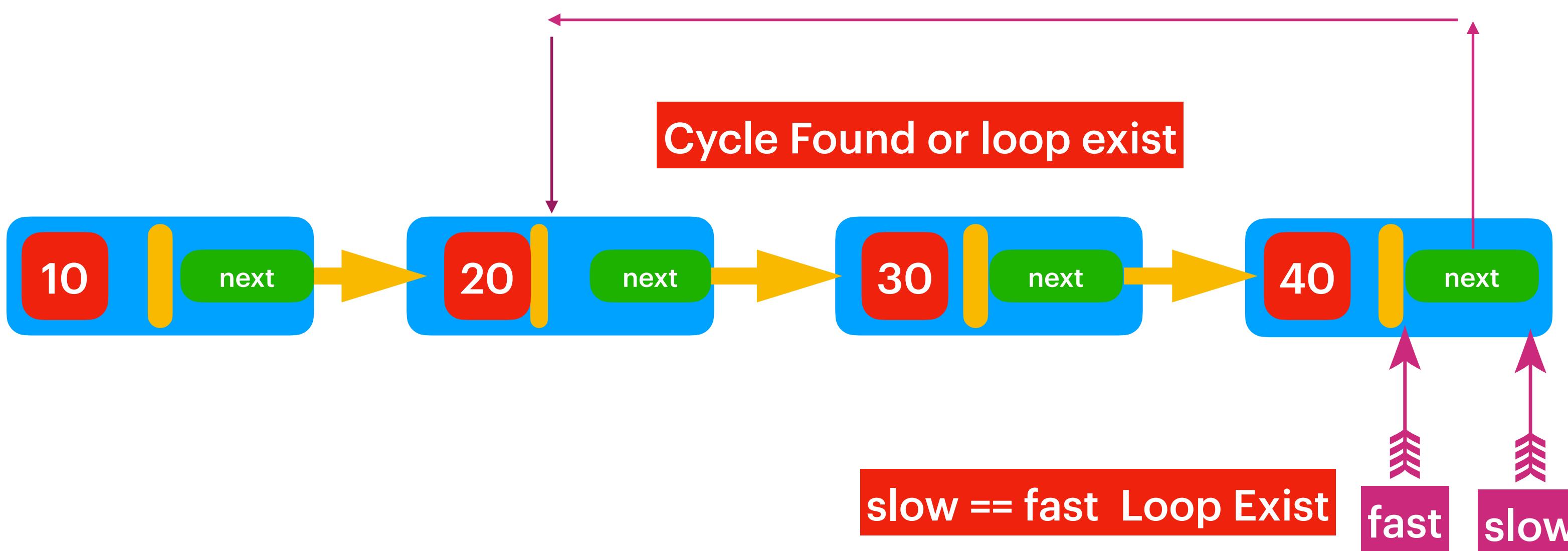
slow pointer jumps 1 step at a time.  
fast pointer jumps 2 steps at a time.



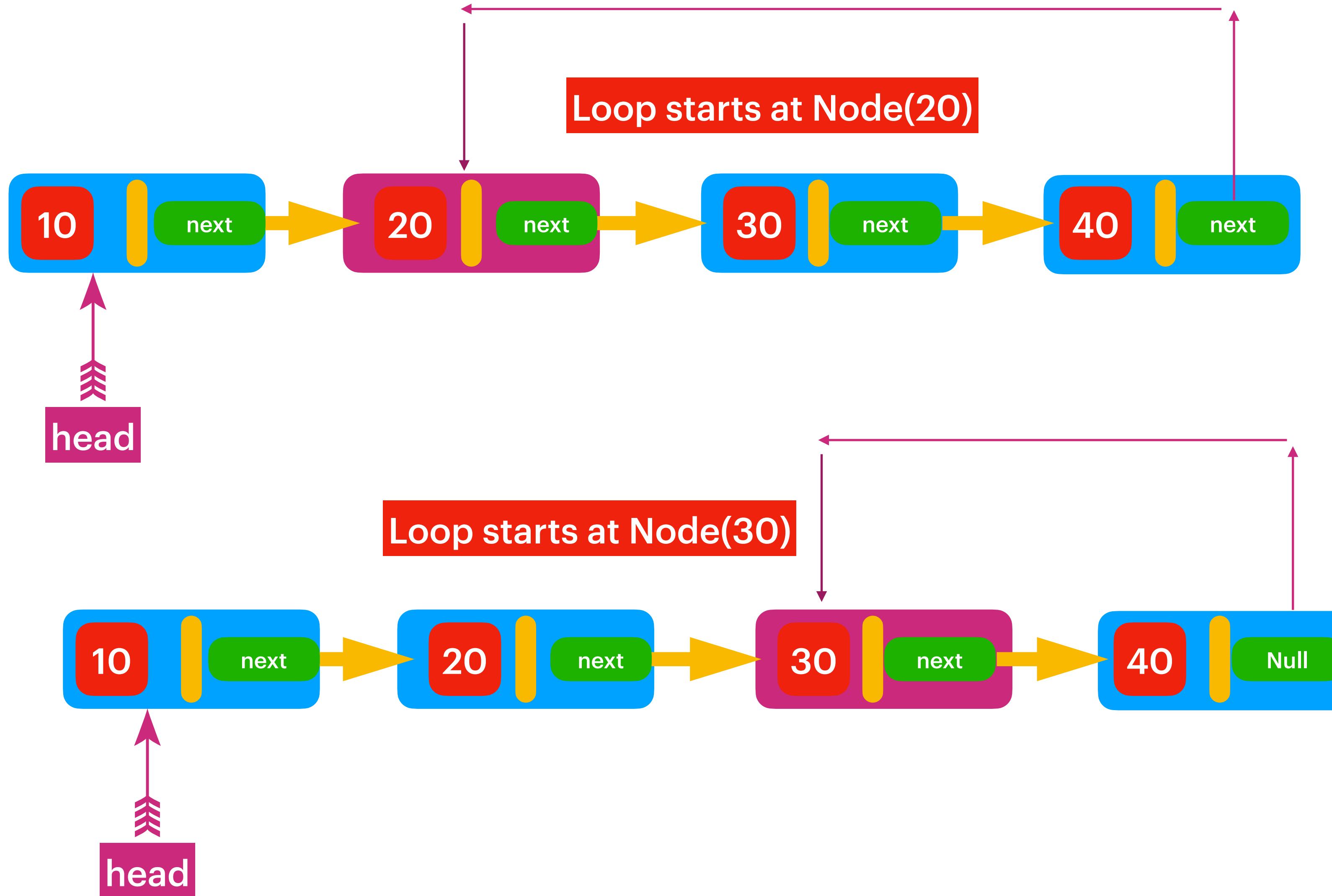
## Detect the Cycle In List



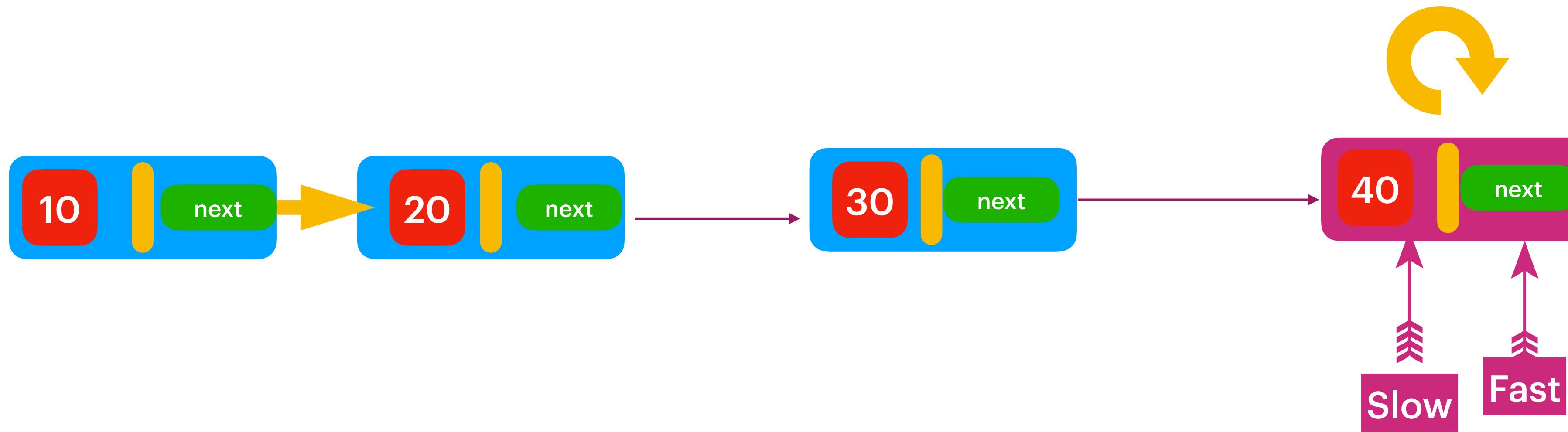
slow pointer jumps 1 step.  
fast pointer jumps 2 steps at a time.

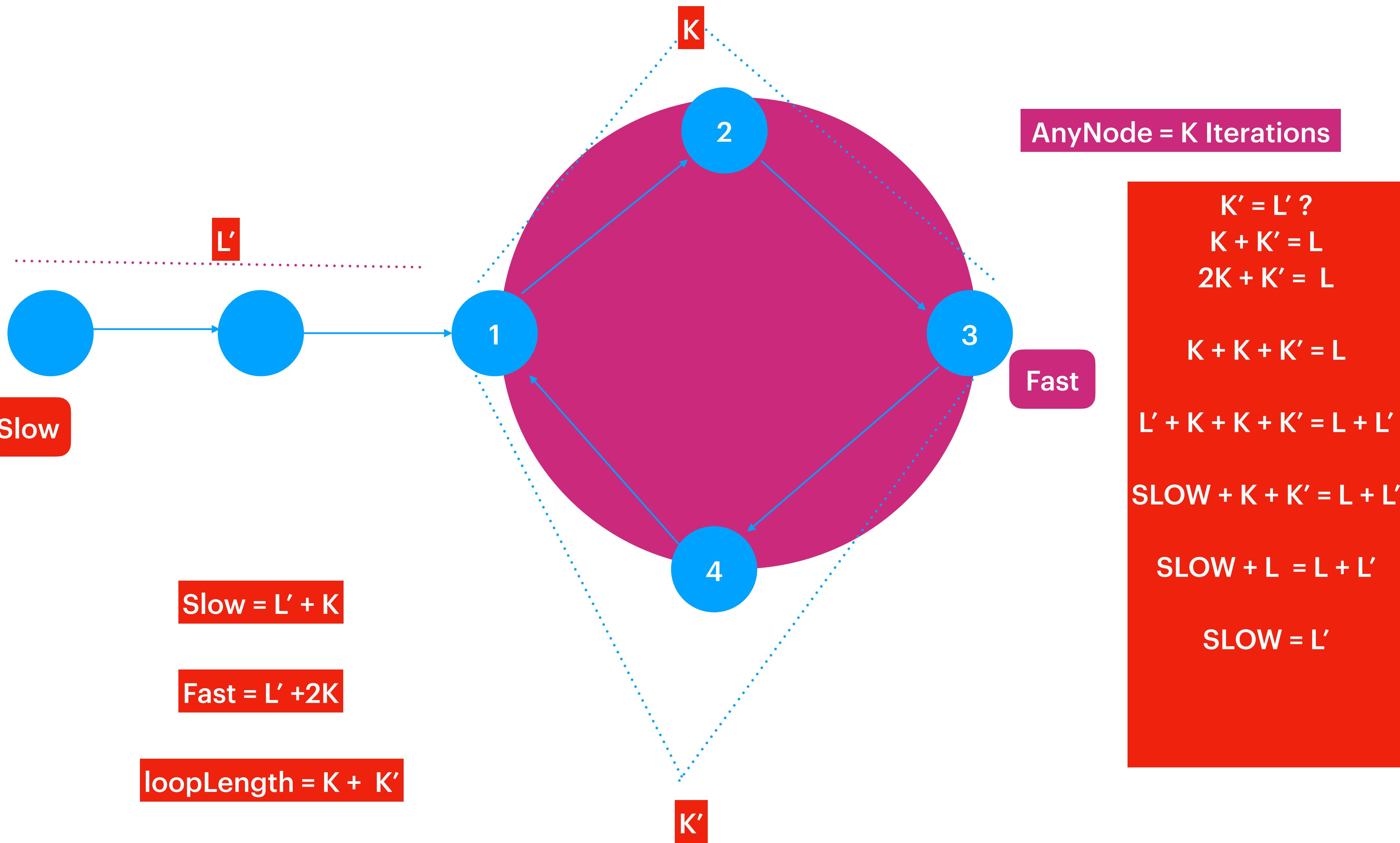


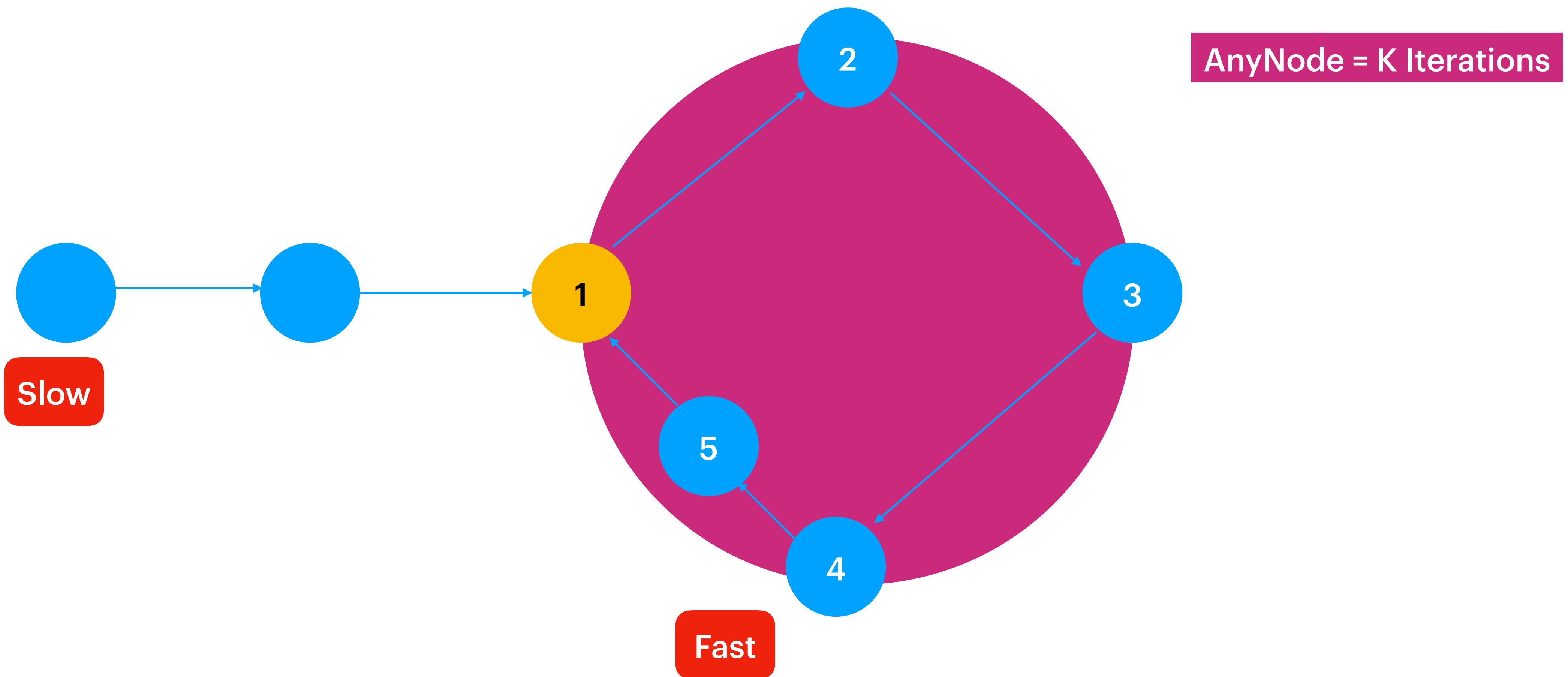
## FindOut Loop Starting point

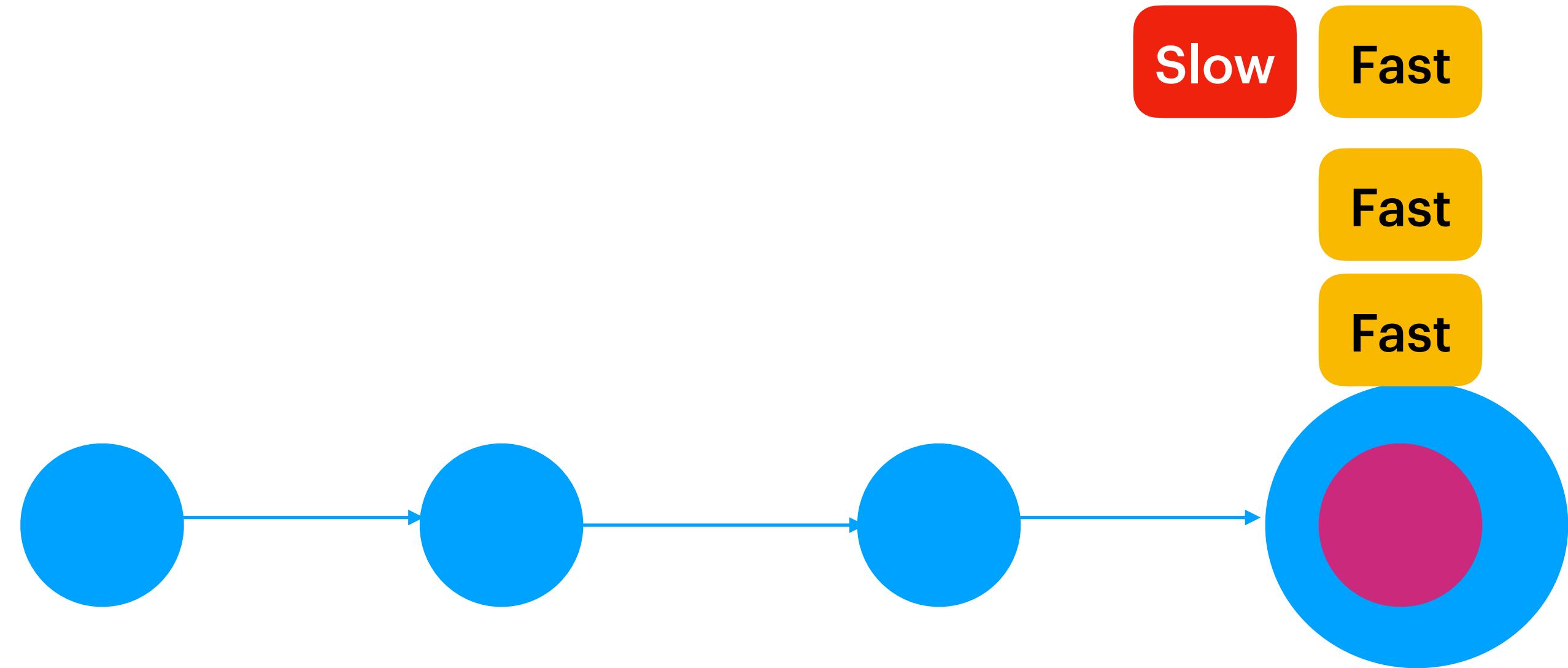


## FindOut Loop Starting point









**Slow =  $L' + K$**

**Fast =  $L' + 2K$**

**loopLength =  $K + K'$**

**$K' = L' ?$**

$$2K + K' = L$$

$$2K + K' = L$$

$$K + K' = 0$$

$$K + 0 = 0$$

$$K = 0$$

$$5K = 0$$

$$10K = 0$$

$$K + K + K' = L$$

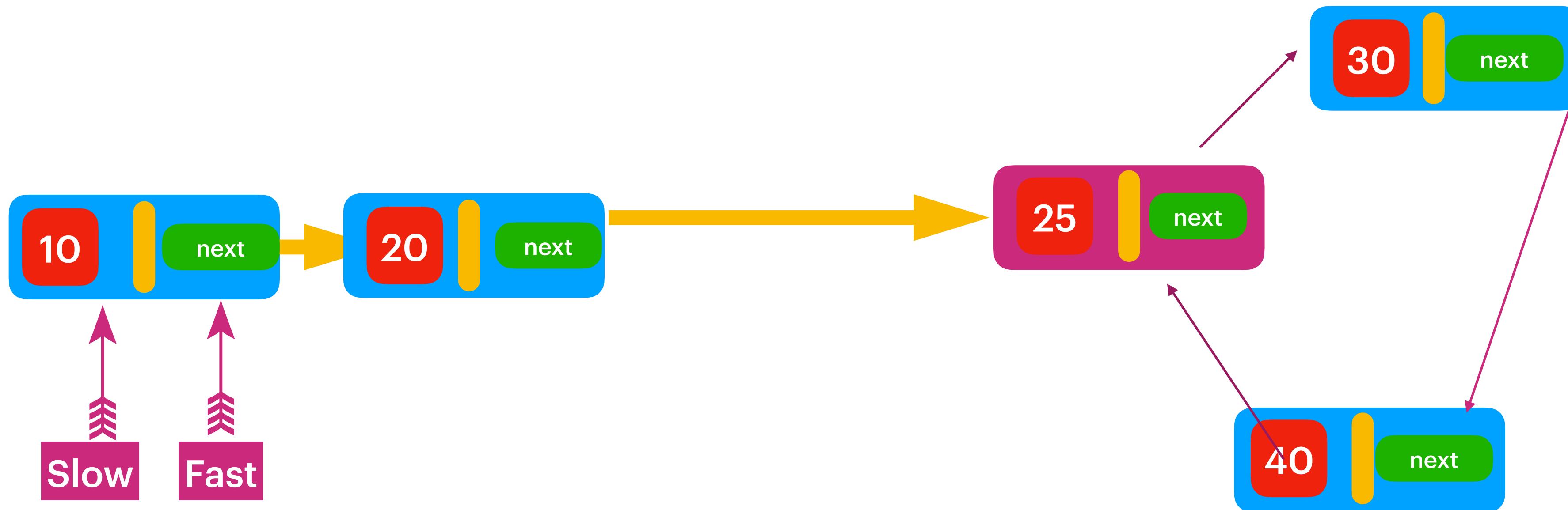
$$L' + K + K + K' = L + L'$$

$$SLOW + K + K' = L + L'$$

$$SLOW + L = L + L'$$

$$SLOW = L'$$

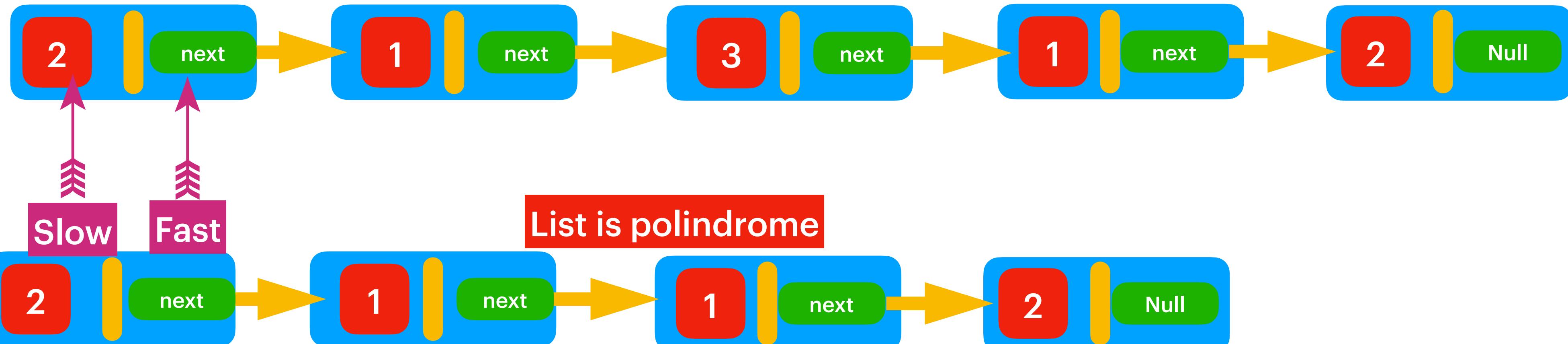
## FindOut Loop Starting point



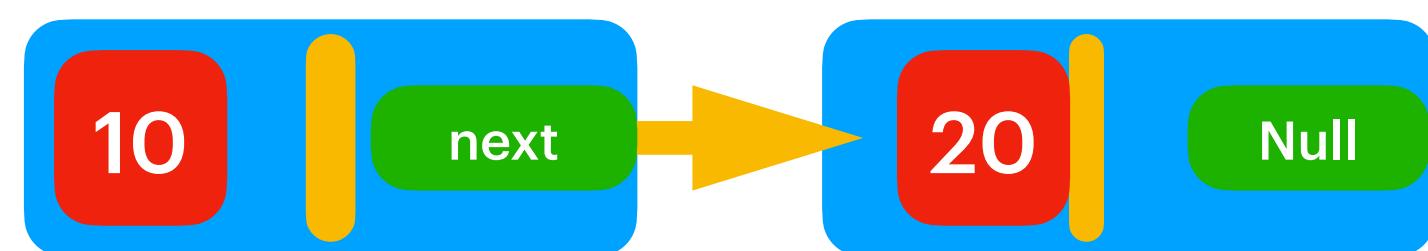
Loop starts at Node(20)

## Palindrome Linked List

List is palindrome

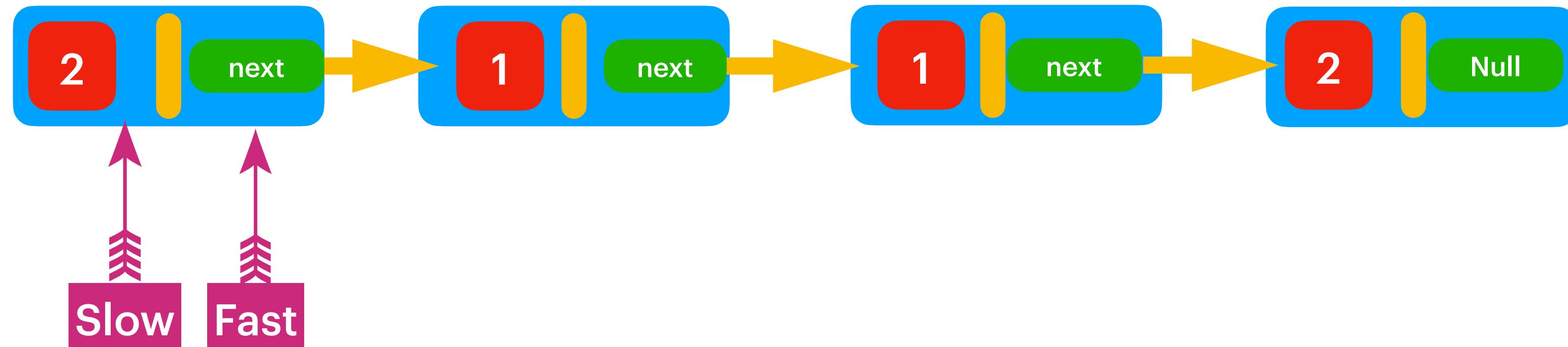


List is not palindrome

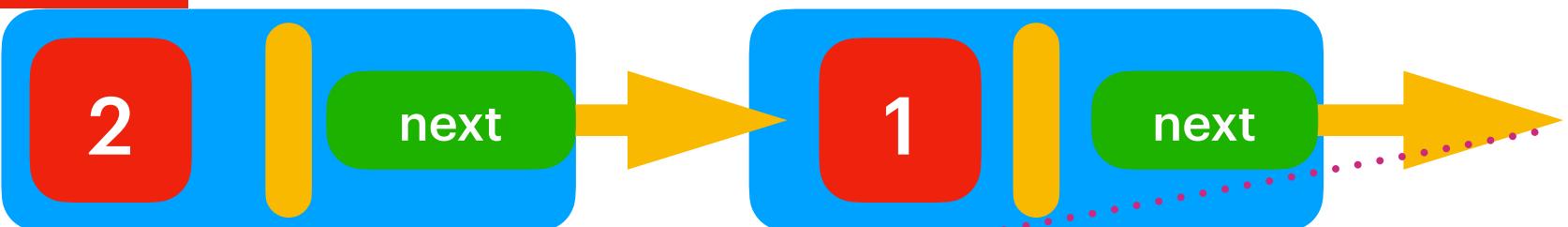


## Palindrome Linked List

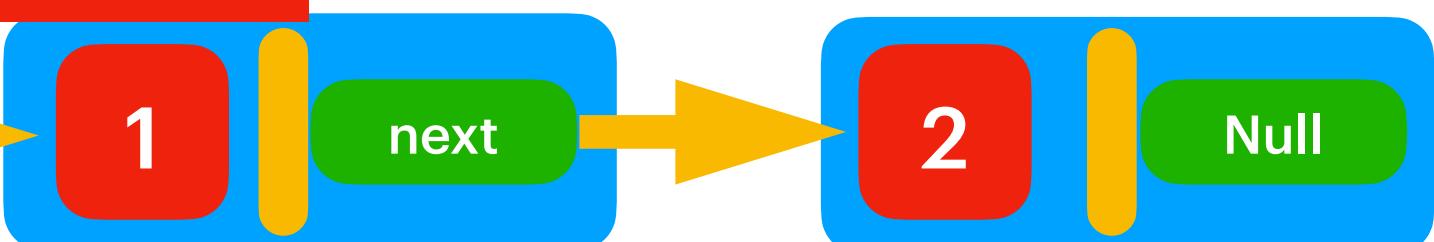
List is palindrome



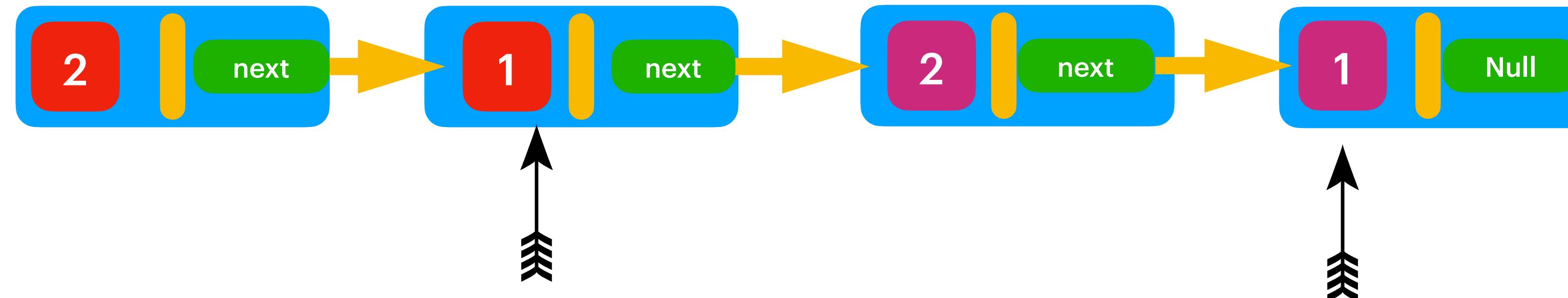
First Half



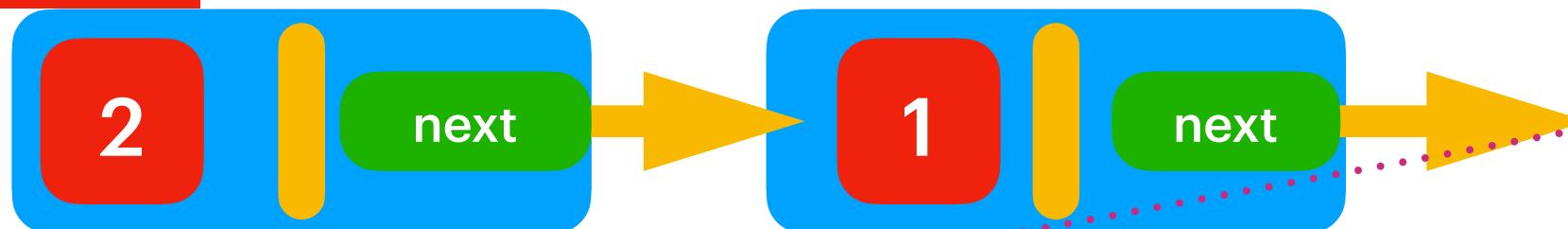
2nd Half



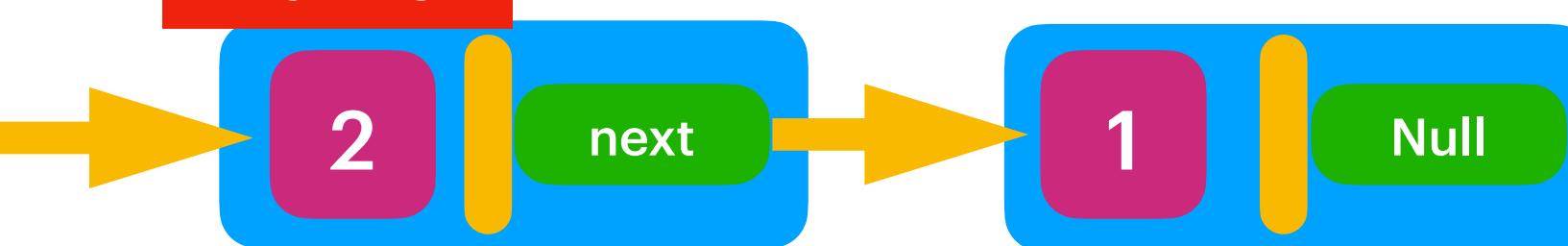
**Reverse the 2ndHalf**



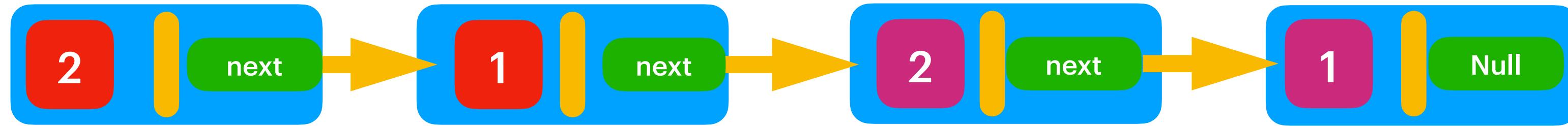
**First Half**



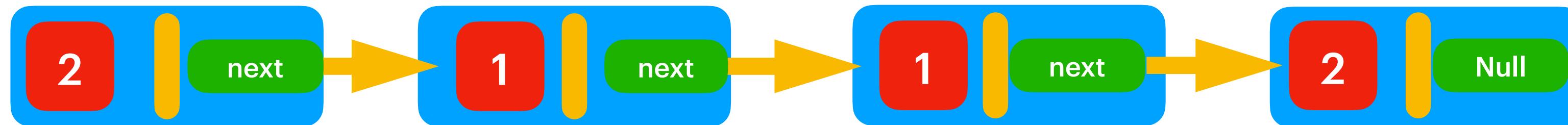
**2nd Half**



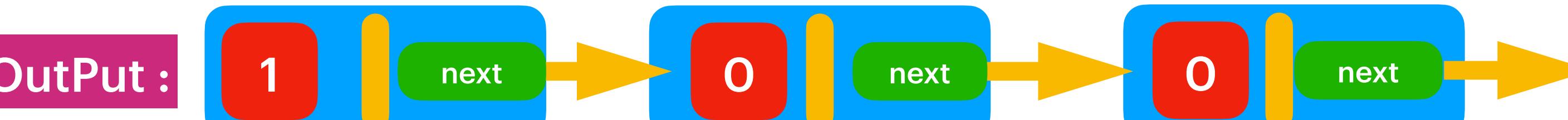
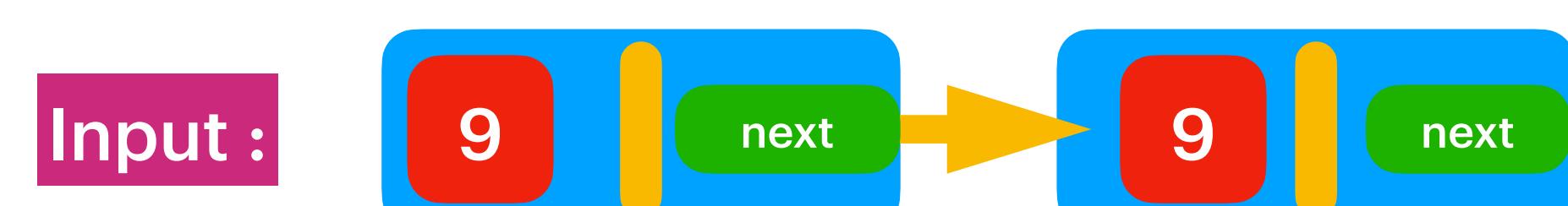
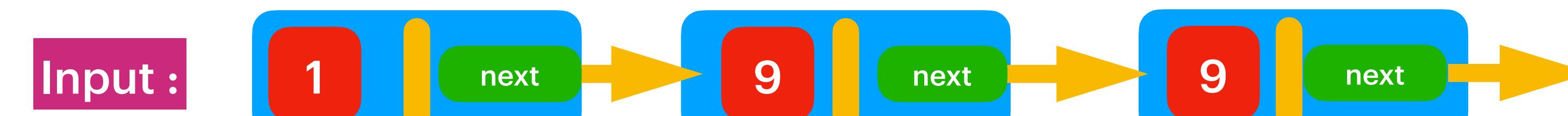
**Compare each node from 1st Half & 2nd Half !!! Check does they are equal**



Reverse the 2ndHalf to get back to original List

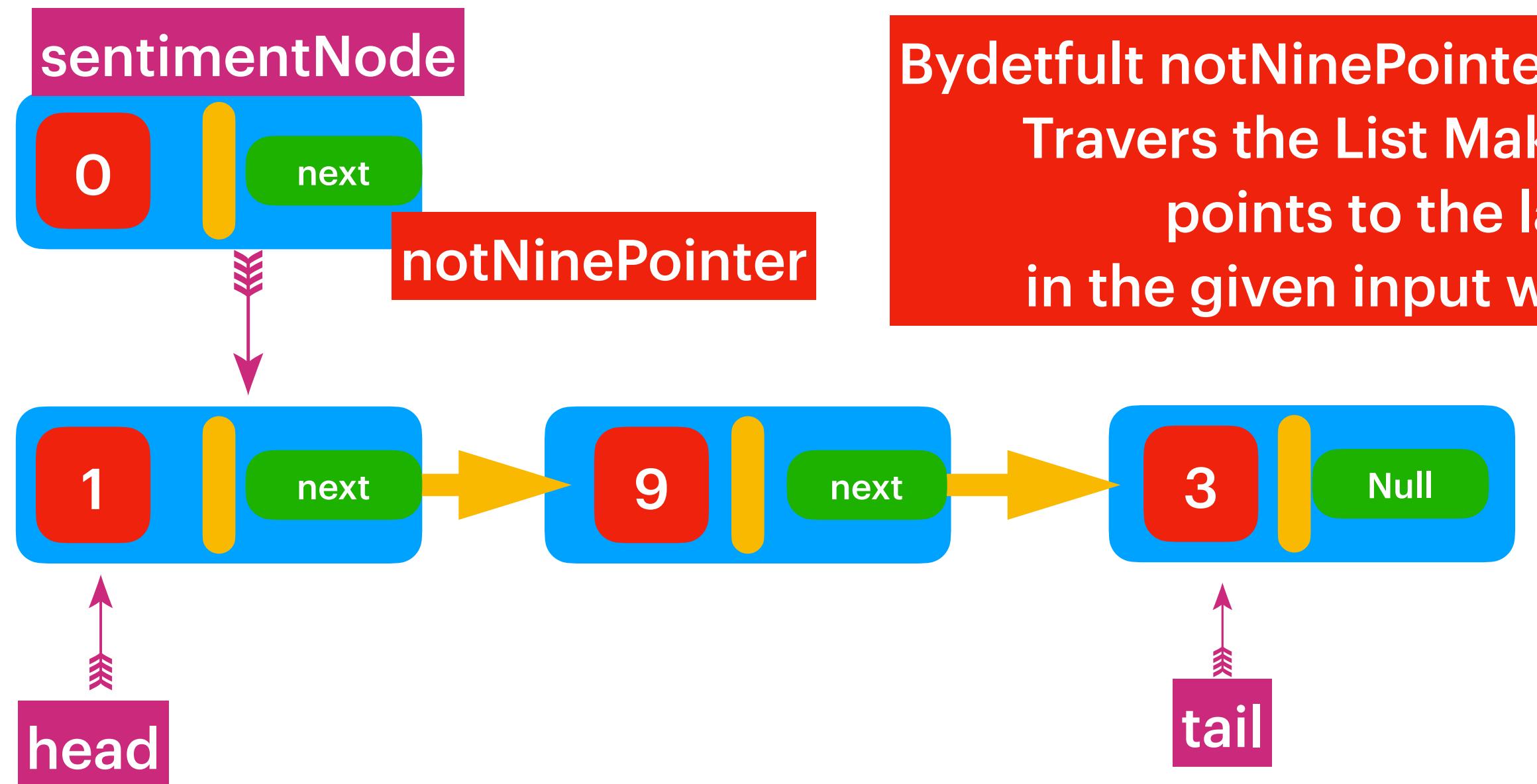


## Plus One to List

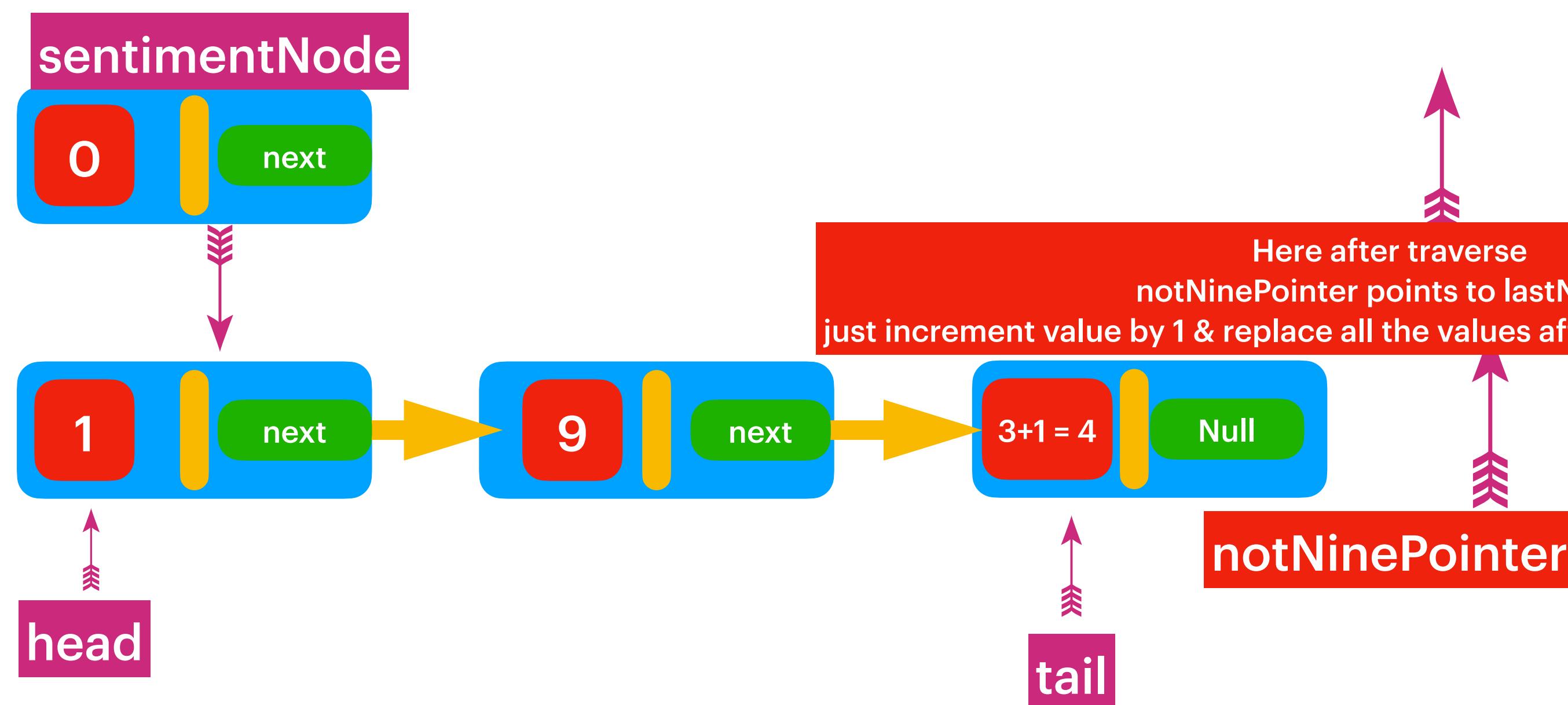


## Plus One to List

Take `sentimentNode`, which holds the value of '0' , mark `sentimentNode` next to head.



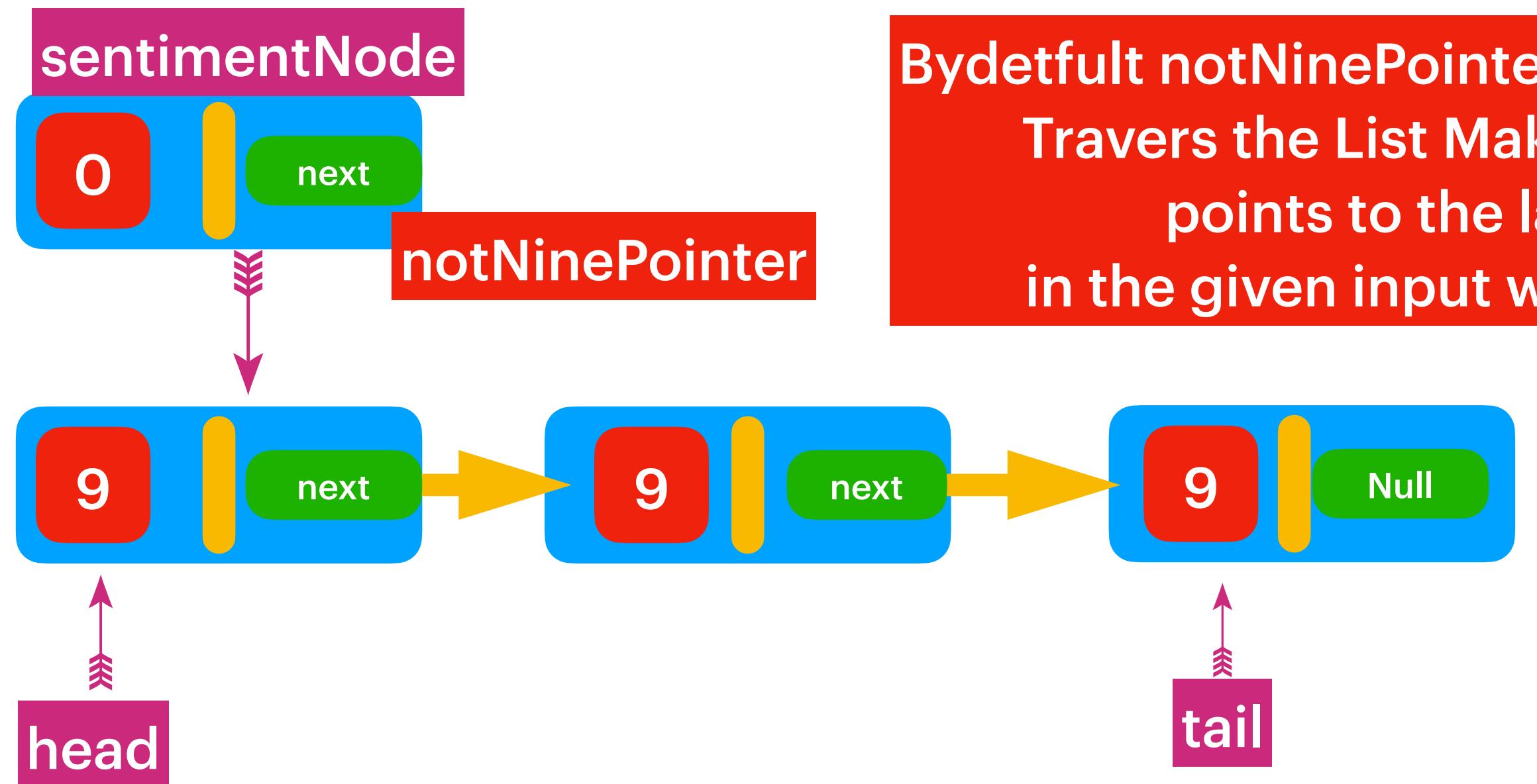
By default `notNinePointer` points to `sentiment Node`,  
Travers the List Make sure, `notNinePointer`  
points to the last possible Node  
in the given input which is not equals to 9.



Key Point : while returning check  
if the `sentimentNode` value is zero or 1,  
If it is zero then return `sentimentNode.next`  
else return `sentimentNode`.

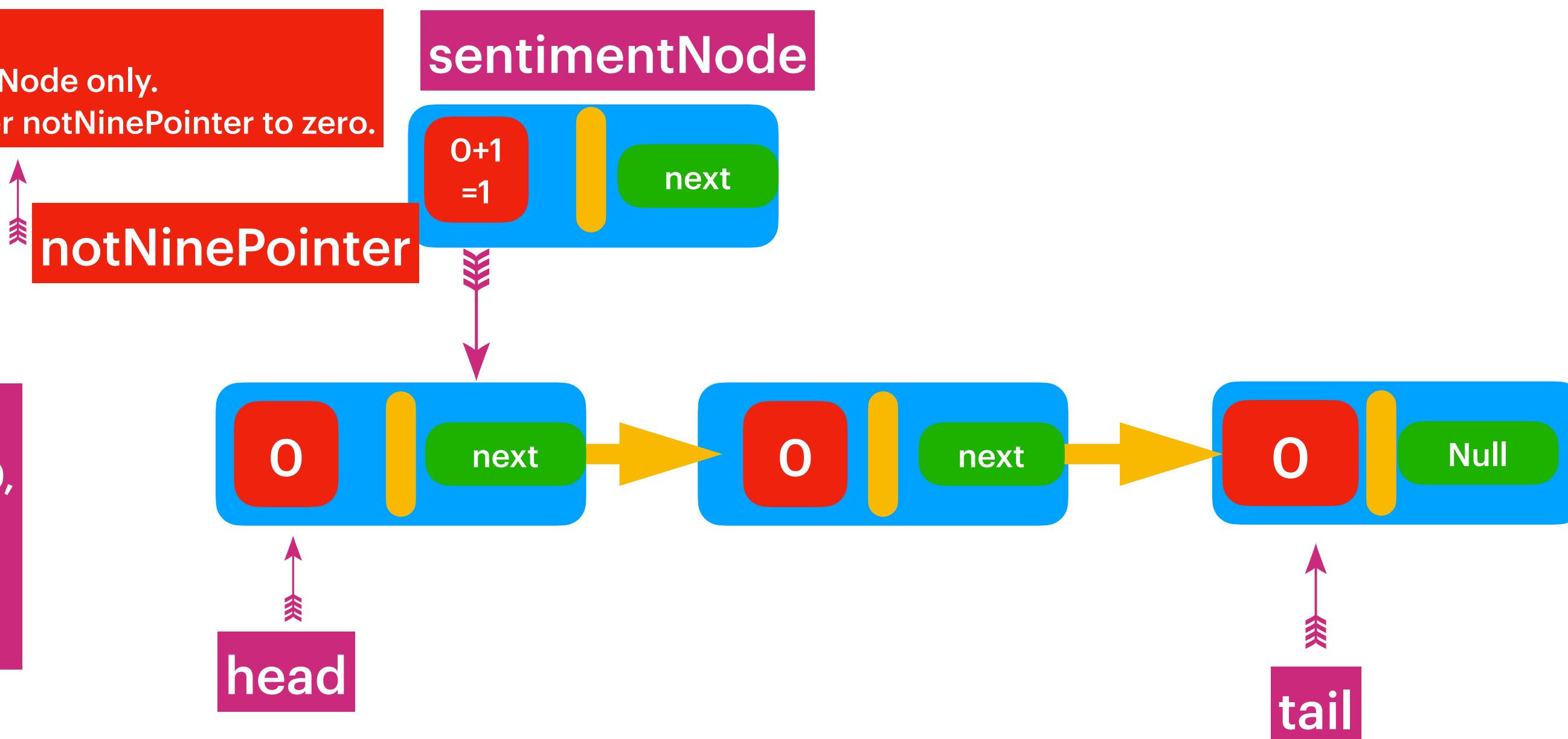
## Plus One to List

Take `sentimentNode`, which holds the value of '0' , mark `sentimentNode` next to head.



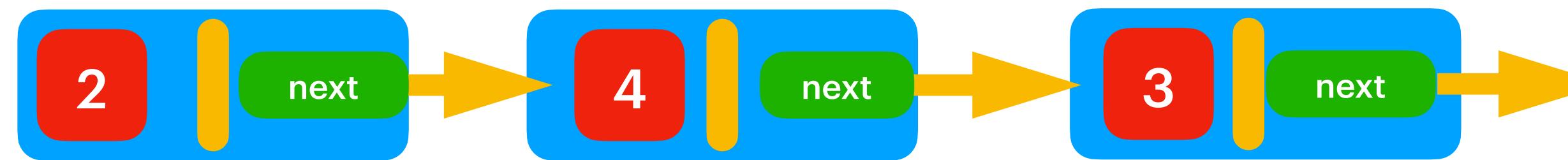
By default `notNinePointer` points to `sentimentNode`,  
Travers the List Make sure, `notNinePointer`  
points to the last possible Node  
in the given input which is not equals to 9.

Here after traverse also  
`notNinePointer` points to `sentimentNode` only.  
just increment by one & replace all the values after `notNinePointer` to zero.

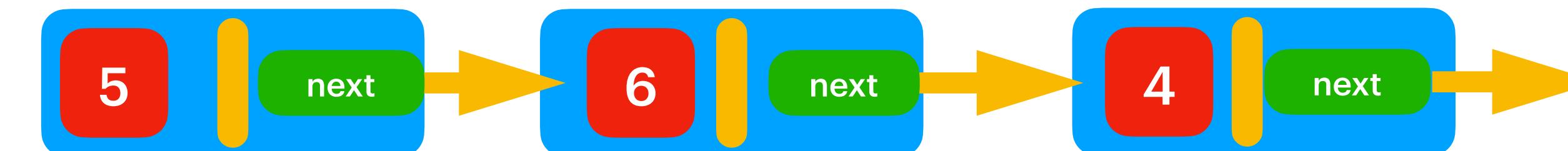


Key Point : while returning check  
if the `sentimentNode` value is 1 or 0,  
If it is 1 then return `sentimentNode`  
else return `sentimentNode.next`

## Adding Numbers :: Inputs are in reverse order !!!



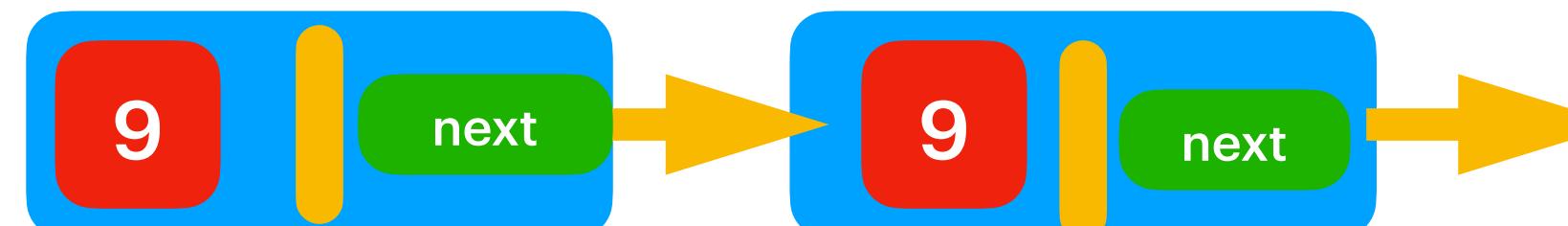
The inputs are in reverse order.  $\Rightarrow 342 + 465 = 807$



Output :



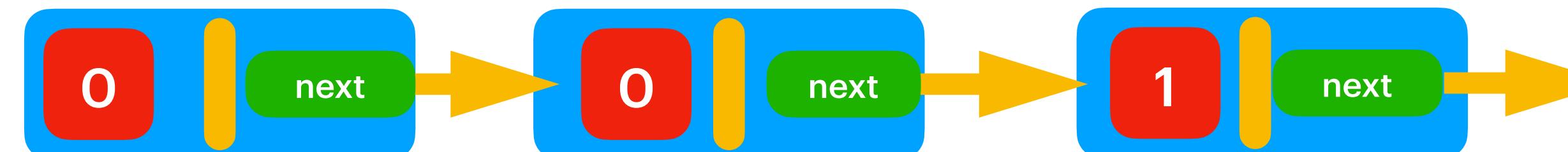
Look at the output in reverse order :



The inputs are in reverse order.  $\Rightarrow 99 + 1 = 100$

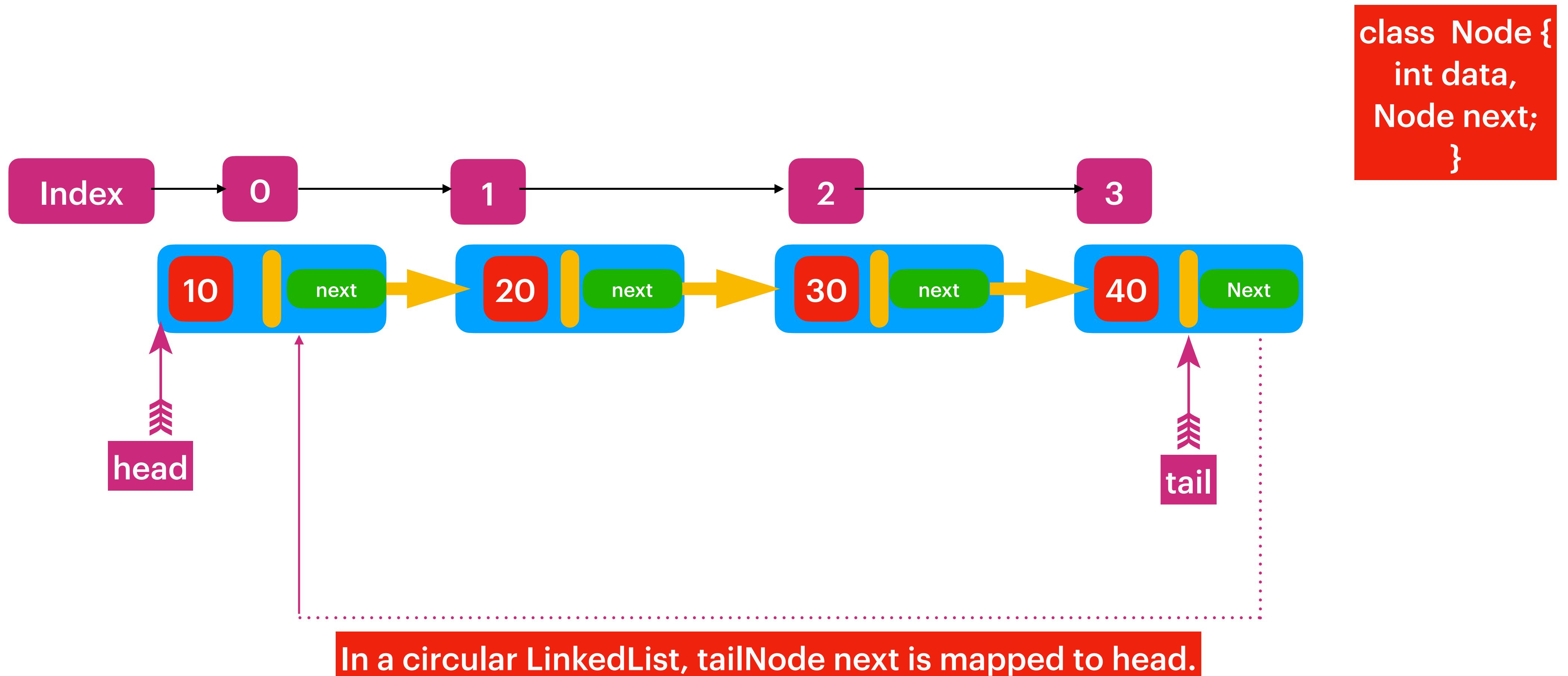


Output :



Look at the output in reverse order :

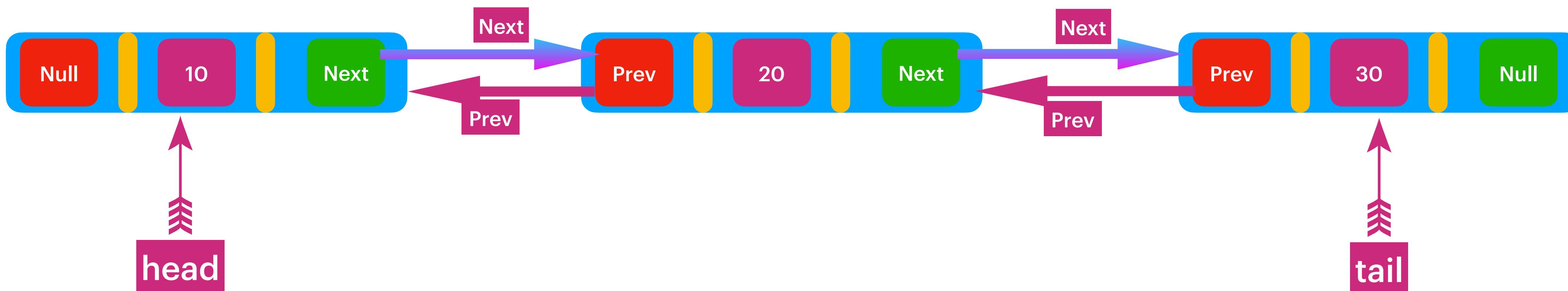
## CircularLinkedList



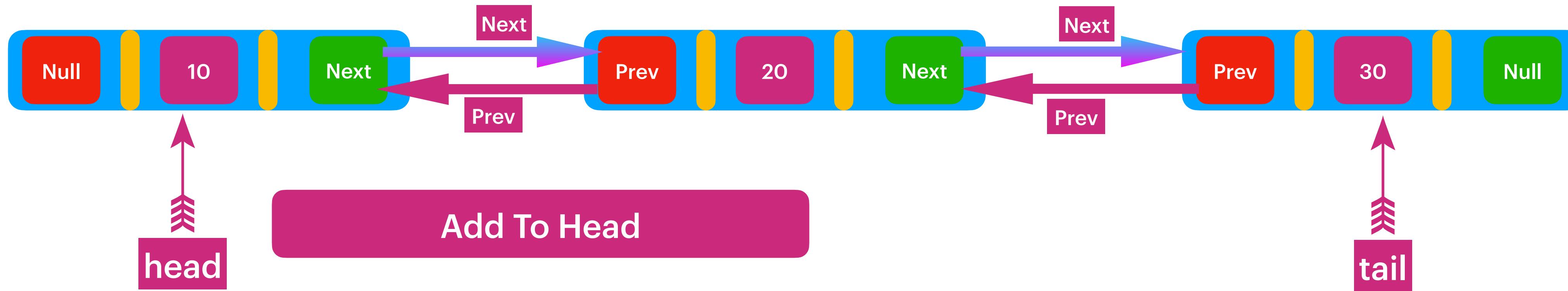
## Double Linked List



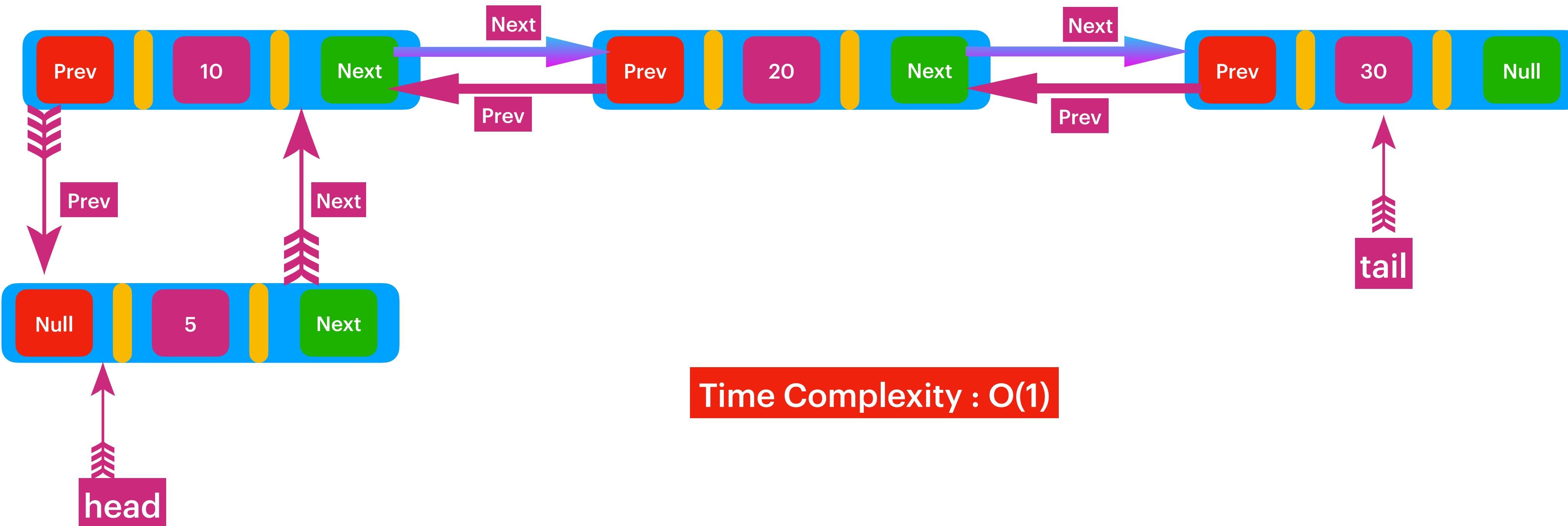
Double Linked List has the reference of nextNode and its previous Node. So that we can traverse both in forward and reverse directions. Double Linked List simply fees the insert & delete operations.



```
class Node {  
    int data,  
    Node next;  
    Node prev;  
}
```



Make newNode next to head & head prev to new Node.  
Mark newNode as head.



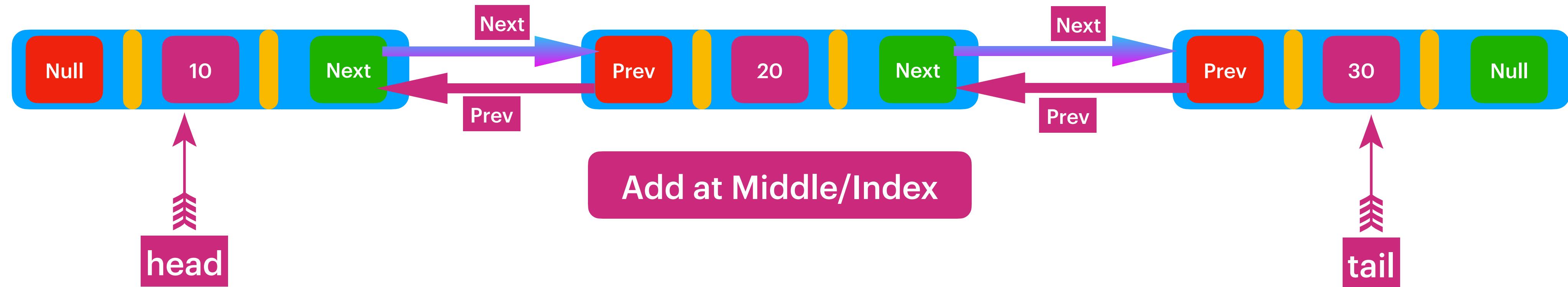


Add To Tail/Last

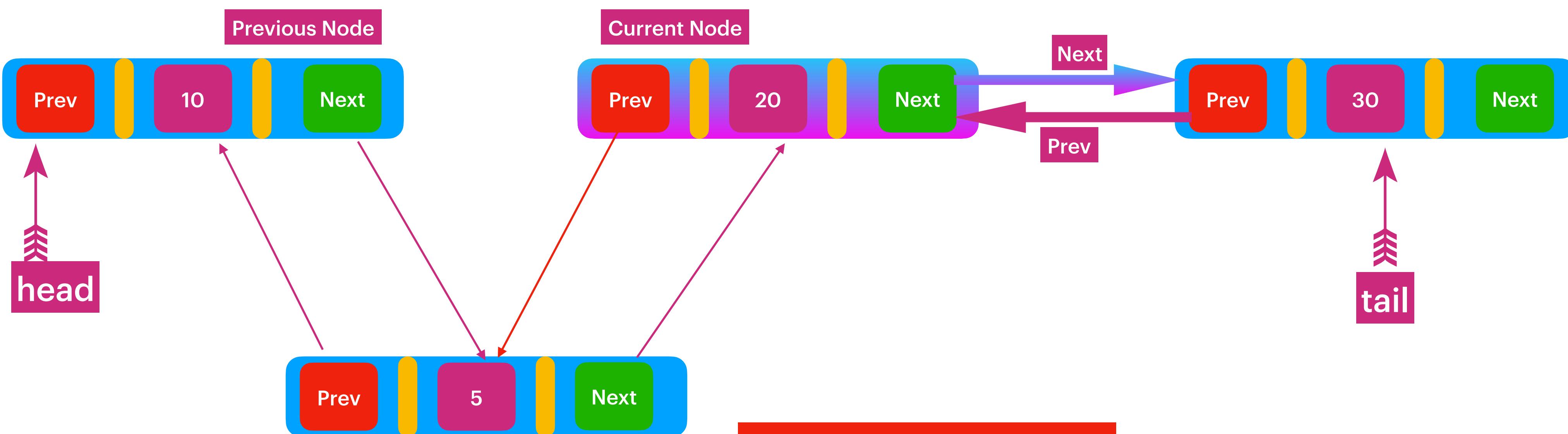
Make newNode prev to tail & tail next to new Node.  
Mark newNode as tail.

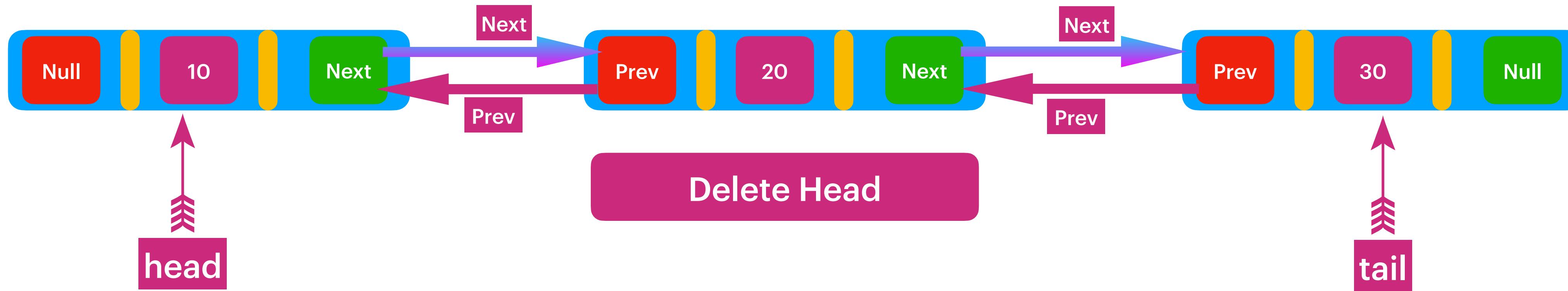


Time Complexity : O(1)

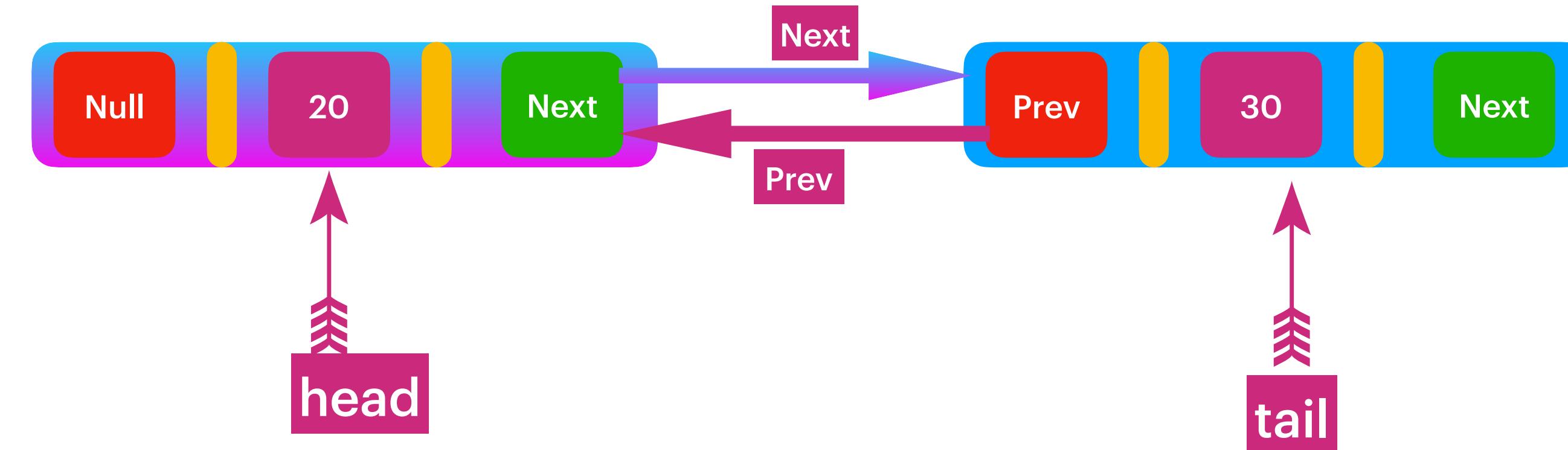


Traverse till currentNode.  
Mark currentNode prev to newNode & previousNode next to newNode.  
Mark newNode prev to previousNode & newNode next to currentNode.

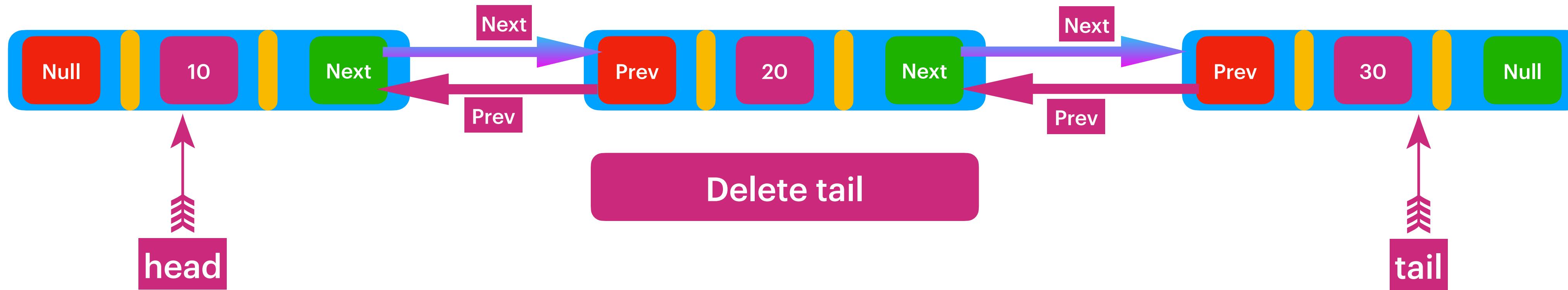




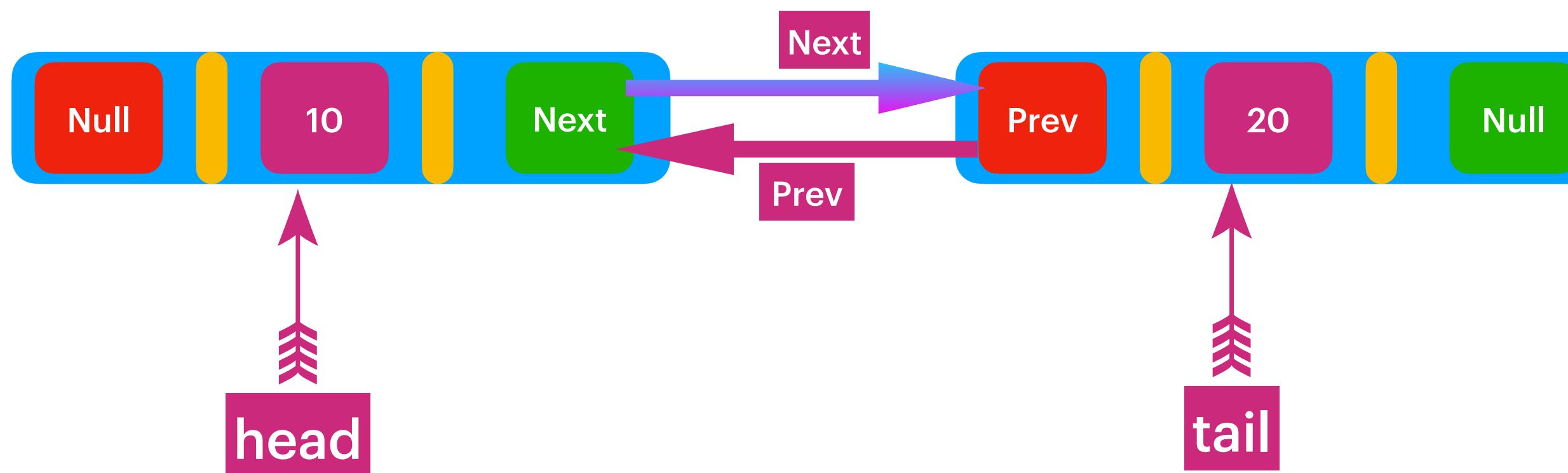
Mark head.next as newHead, then make newHead prev to null.



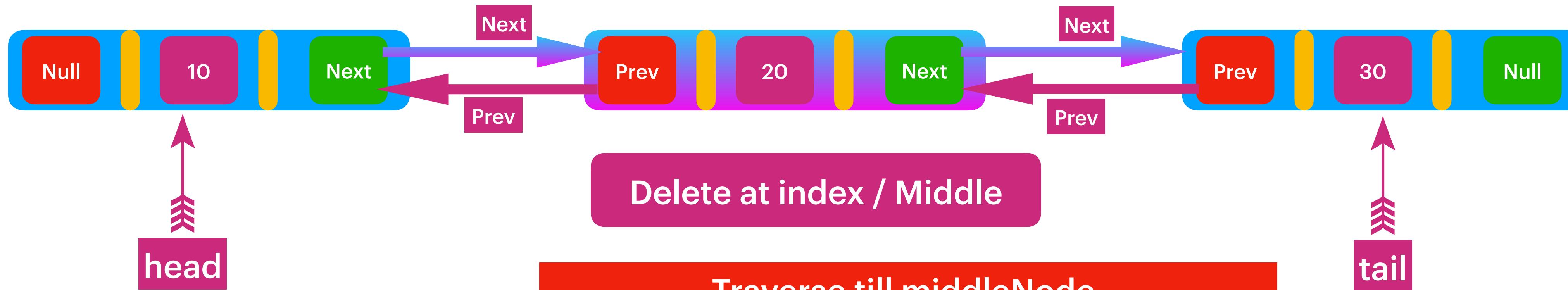
Time Complexity : O(1)



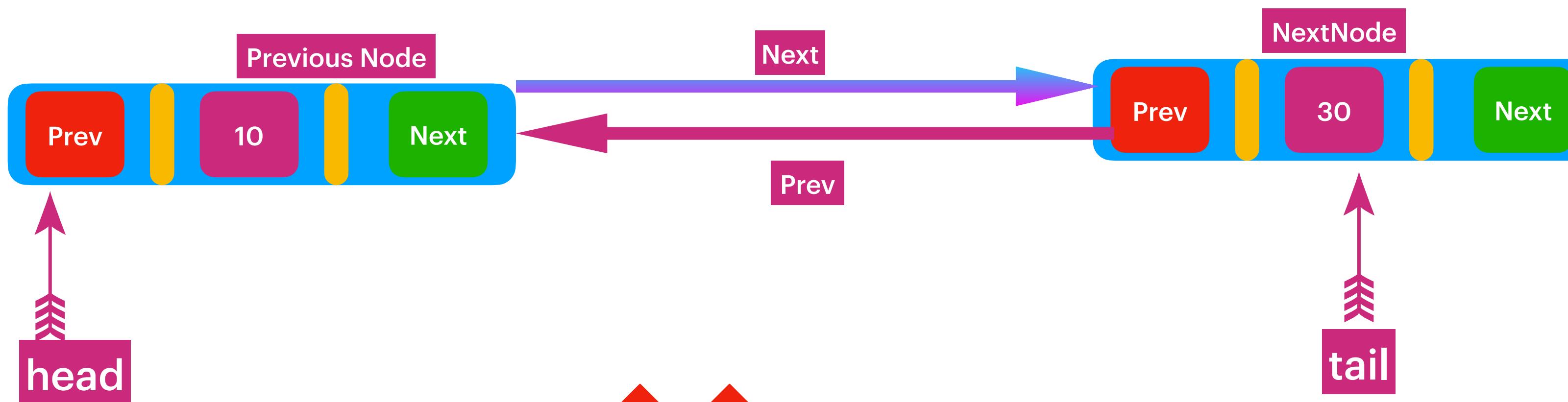
Mark tail.prev as new tail, then make new tail next as null.



TimeComplexity : O(1)



Traverse till middleNode.  
 Mark MiddleNode next to previousNode next.  
 Mark MiddleNode prev to nextNode prev.  
 Mark MiddleNode next & prev as null.



**TimeComplexity : O(n)**



Now no reference is pointing to middleNode so that middleNode will be garbage collected.

## ArrayList

**Advantage : Accessibility on index O(1)**

**Disadvantages :**

**Deletion/Insertion : O(n) time / shift Operation**

**Search : O(n)**

**Allows duplicates.**

## LinkedList

**Advantage**

**Deletion a Node: O(n) time & O(1) shift Operation**

**Insertion Node : O(1)**

**Disadvantages :**

**Accessibility on index O(n)**

**Search : O(n)**

**Allows duplicates.**

**Search & Insertion should be done on constant time .**

**Should allow only unique elements.**

## Hashing Data Structures

Hashing is been introduced to improve the searching capability & avoiding the duplicates.

On Hashing we can search the element in Constant time  $O(1)$  in average case.

In worst case it would be taking  $O(\log n)$ .

Set

Set allows only value.  
Does not allow duplicates.

Map

Map allows Key, value Pairs.  
Allows only unique keys.  
Allows Duplicate values.  
Map replaces the value when  
the key is present.

## How Hashing works ?

Simple, in hashing we try to understand the behaviour of a data then we derive a hash logic so that we can link similar group of data.

Ultimately this Hash Logic makes an important role here.

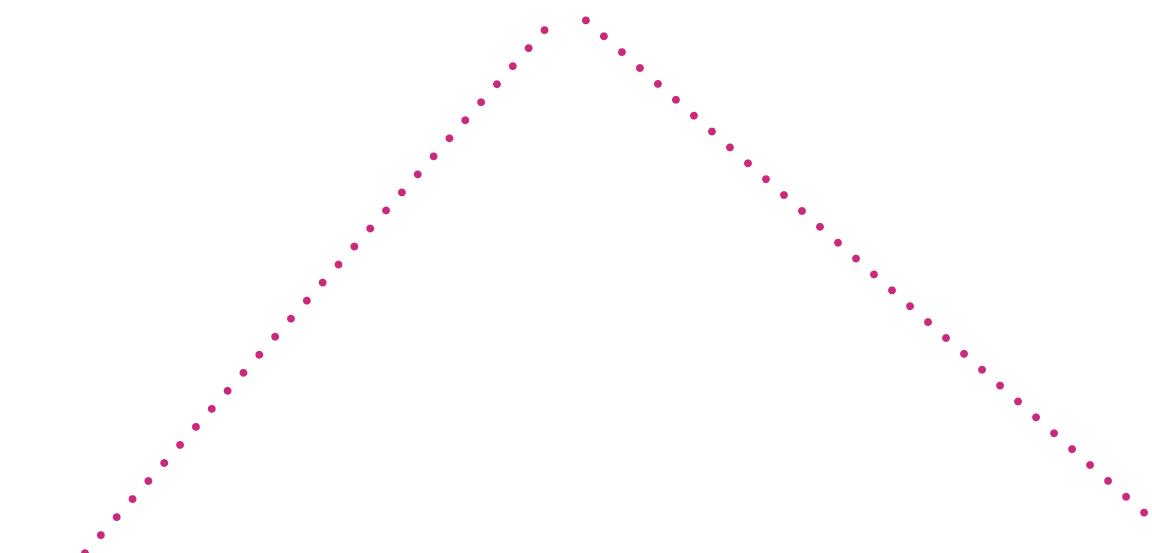
In hashing similar group of data can be represented by Bucket.

In simple first we decide, how many buckets we need, then we try to link data to buckets based on their hash logic.

**Hash Collisions :** There is always high possibility that different sets of data , can point to same bucket, it does not mean that they are equal.

This is called Hash Collision, to support this we map LinkedList to the bucket.

## Hashing Data Structures



### Set

Set allows only value.  
Does not allow duplicates.

### Map

Map allows Key, value Pairs.  
Allows only unique keys.  
Allows Duplicate values.  
Map replaces the value when the key is present.

Finally, in hashing data structure  
First we understand the behaviour of a data, we decide how many initial buckets we need then derive the hash logic, this hash logic is going to represent a bucket.  
As we know that there is a hash collision, while adding element to avoid duplicates we compare then add .

Hash Logic = element % 5

10%5 = Bucket[0]

14%5 = Bucket[4]

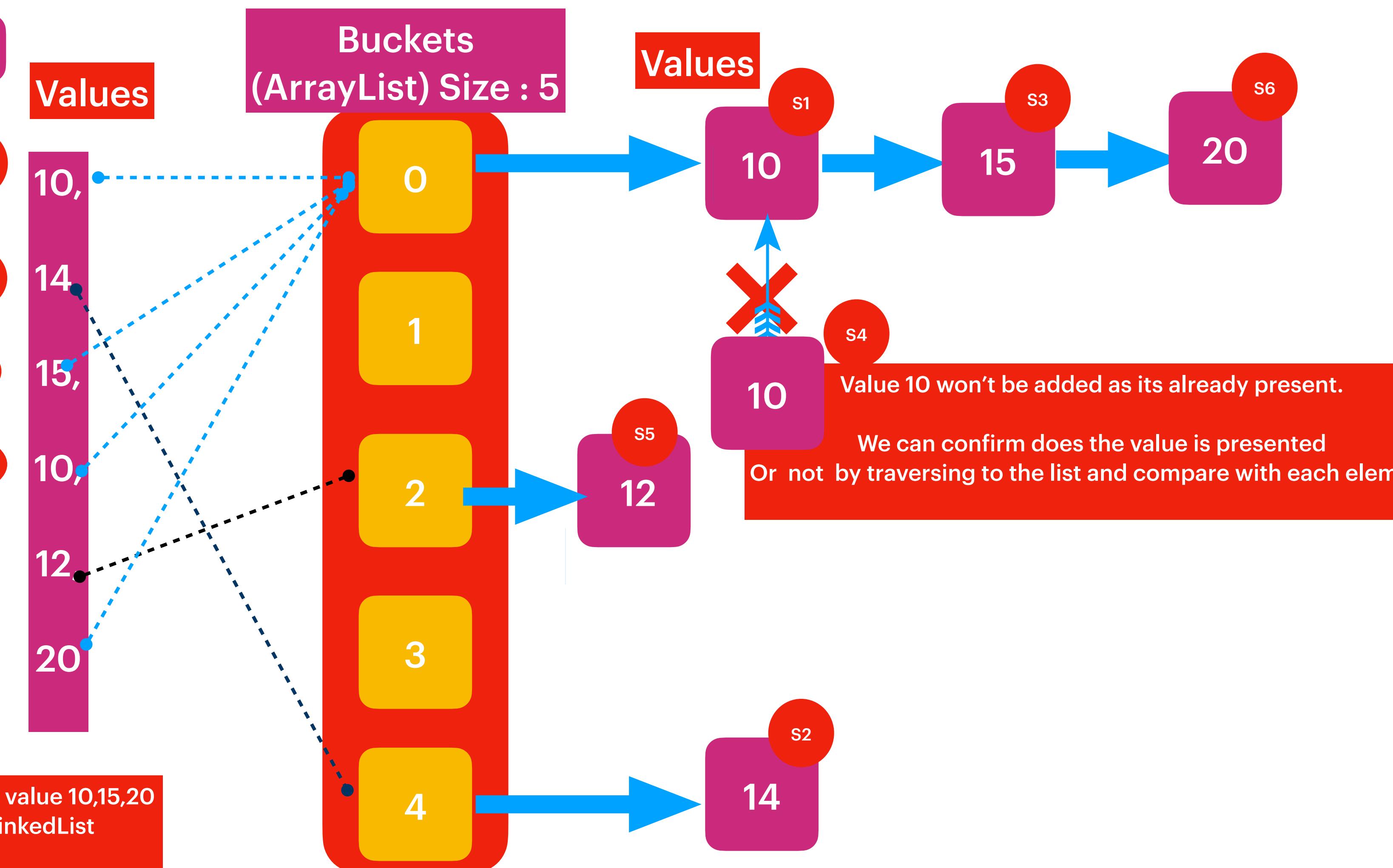
15%5 = Bucket[0]

10%5 = Bucket[0]

12%5 = Bucket[2]

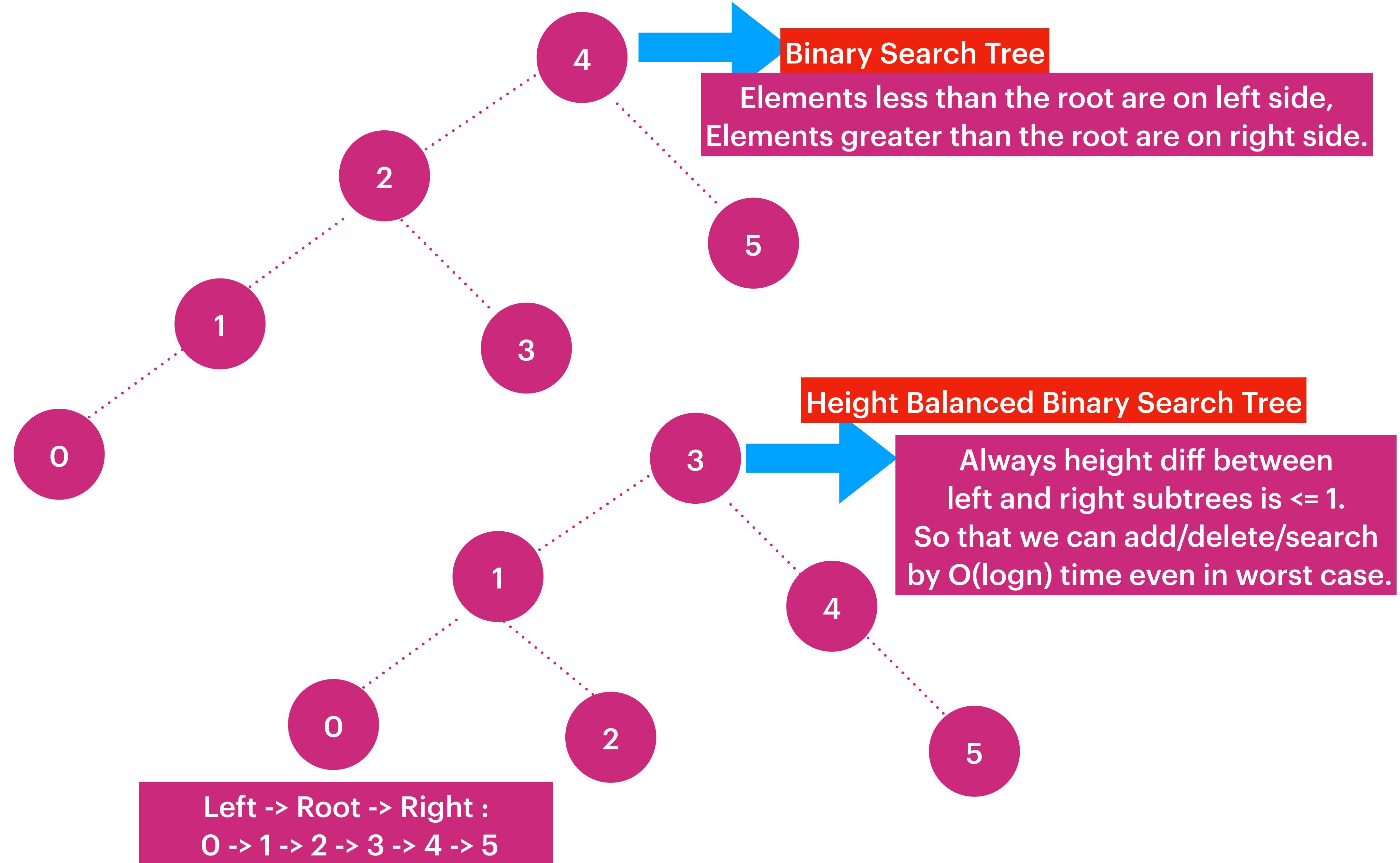
20%5 = Bucket[0]

Observe there is a hash collision at value 10,15,20  
so that we addressed using LinkedList



A bad hash logic or bad dataset can result to pointing of all the elements to single bucket. To address this after reaching threshold limit we would convert LinkedList to Height based Balanced Binary Search Tree (AVL), which enables us to add/search/delete element in O(logn) time.

**Time Complexity :**  
Insertion Node : Avg Case O(1) / Worst Case O(logn)  
Delete Node : Avg Case O(1) / Worst Case O(logn)  
Search : Avg Case O(1) / Worst Case O(logn)



## Hashing Data Structures

List

ArrayList

LinkedList

Set

**Set allows only value.  
Does not allow duplicates.**

Map

**Map allows Key, value Pairs.  
Allows only unique keys.  
Allows Duplicate values.  
Map replaces the value when  
the key is present.**

Hash Logic = element % size

Initial bucketSize = 5

LoadFactor = 0.75

6 => 6%5 => Bucket(1)

7 => Bucket(1)

8 => Bucket(3)

9 => Bucket(4)



Reached 75% capacity  
= Math.round(5 \* 0.75) = 4



add(11)

Rehash all the elements



bucketSize = size \* 2 = 10

6%10 = Bucket(6)

7%10 = Bucket(7)

8%10 = Bucket(8)

9%10 = Bucket(9)

11%10 = Bucket(1)

## Importance Of LoadFactor In Hashing

LoadFactor can speak about when to rehash the Data Structure.

Usually we setup LoadFactor to 75%.

It means as and when 75% of the buckets were occupied  
then we double the bucket size.

Rehashing of each element is needed as and when we double the bucket size.

Rehashing is always costly operation so that choosing bucket size will always gives  
best performance.

As per documentation => size = actualSize/loadFactor ,

Assume as per your requirement you need 5 buckets then  
size = Math.round(5/0.75) = 7

### From Documentation :

An instance of `HashMap` has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the `HashMap` class, including `get` and `put`). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

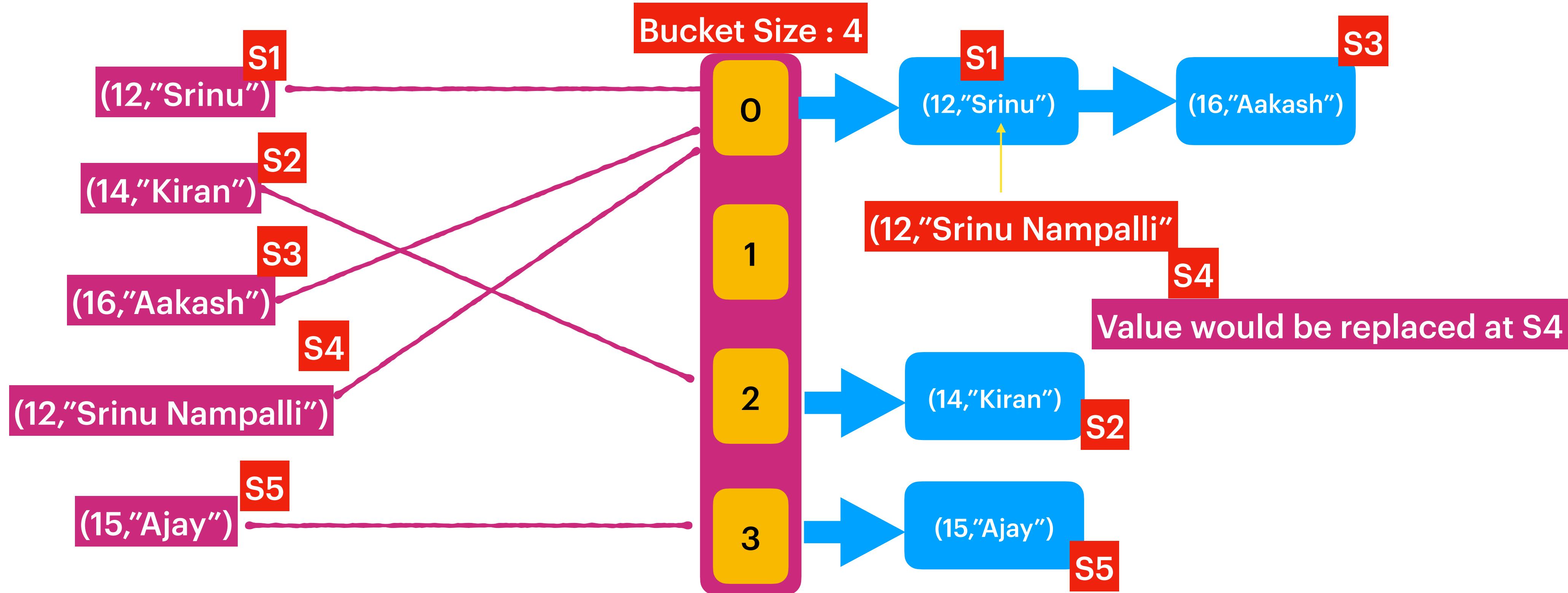
Hash Logic : key % 4

Map Design :

Map allows key & value pairs.

Allows unique keys & duplicate values.

If the key is present the replaces the value.



```
PairNode p1 = new PairNode(12,"Srinu");
```

```
PairNode p2 = new PairNode(12,"Srinu Nampalli");
```

P1

12, "Srinu"

P2

12, "Srinu Nampalli"

p1.equals(p2) => p1Addr == p2Addr always returns false

p1.equals(p2) => p1Addr == p2Addr always returns false

```
p1.equals(p2)  
public boolean equals(Object obj)  
{  
    PairNode p = (PairNode) obj;  
    return this.key.equals(p.key);  
}
```

```
public int hashCode()  
{  
    return key;  
}
```

## Hashing Data Structure

Avoid duplicates .  
Search/add/delete/update : O(1) / O(logn)

Set of add(element) internally uses  
map only =>  
map.put(element, element)

Set

HashSet

Insertion Order  
not maintained

LinkedHashSet

Insertion Order  
maintained

Elements  
will be sorted in  
Ascending  
order.

TreeSet

HashMap

Insertion Order  
not maintained

Map

LinkedHashMap

Insertion Order  
maintained

Elements will be  
sorted in  
Ascending  
order based on  
Key.

TreeMap

## Remove Duplicates

Array not ordered & unsorted Array

Input : {15,1,12,12,15}  
Output : {15,1,12}

Input : {10,10,10}  
Output : {10}

## Single Number

Given a non-empty array of integers `nums`, every element appears twice except for one. Find that single one.  
You must implement a solution with a linear runtime complexity. (\*Use Constant Space)

Input: `nums = [2,2,1]`  
Output: 1

Input: `nums = [4,1,2,1,2]`  
Output: 4

Input: `nums = [1]`  
Output: 1

## Two Sum

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

**Input:** `nums` = [2,7,11,15], `target` = 9

**Output:** [0,1]

**Output:** Because `nums[0] + nums[1] == 9`, we return [0, 1].

**Input:** `nums` = [3,2,4], `target` = 6

**Output:** [1,2]

**Input:** `nums` = [3,3], `target` = 6

**Output:** [0,1]

## Isomorphic Strings

Given two strings s and t, determine if they are isomorphic.

Two strings s and t are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

Input: s = "egg", t = "add"  
Output: true

Input: s = "foo", t = "bar"  
Output: false

Input: s = "paper", t = "title"  
Output: true

s & t are Isomorphic

If and Only If each character in s can map to each character in t uniquely

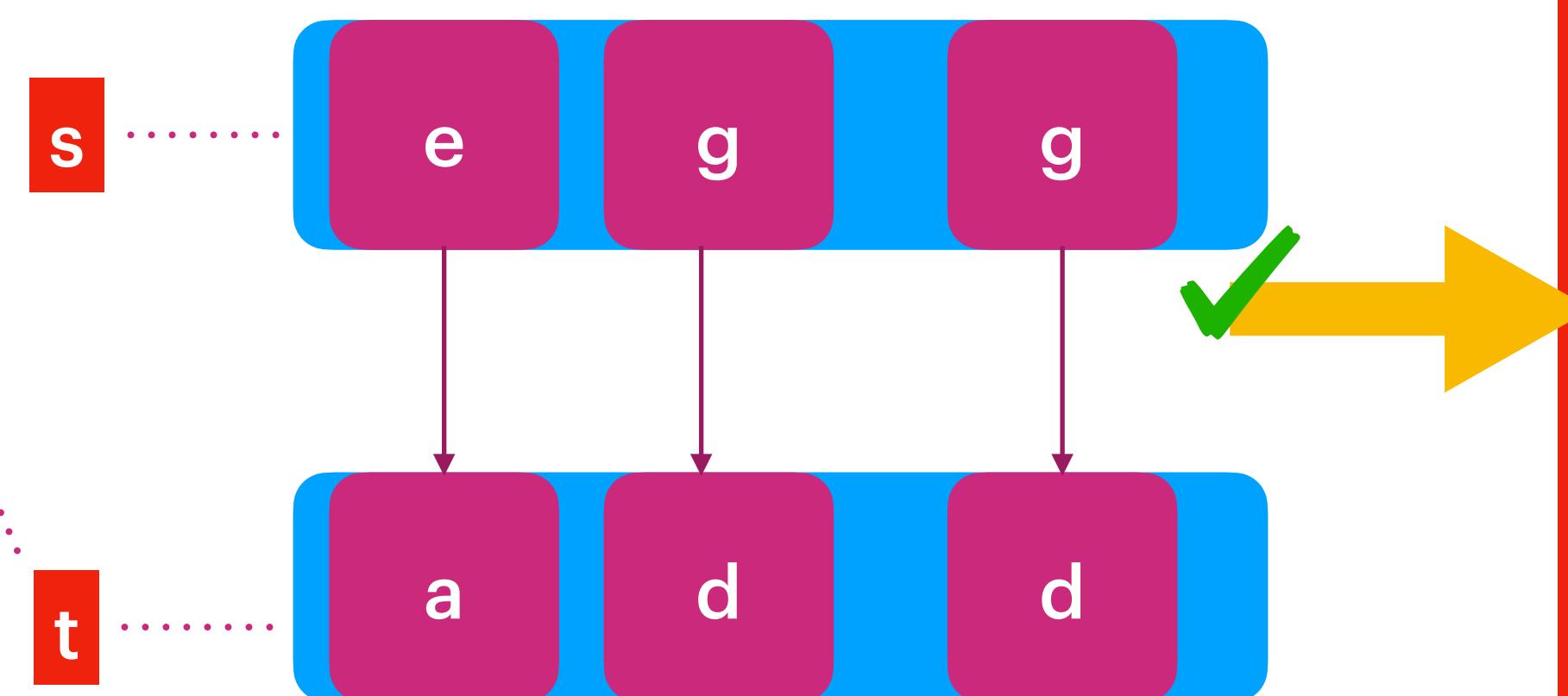
In simple single character in 's' can not be mapped to multiple character in 't' and also single character in 't' can not be mapped multiple characters in 's'.

Input: s = "egg", t = "add"

Output: true

Input: s = "foo", t = "bar"

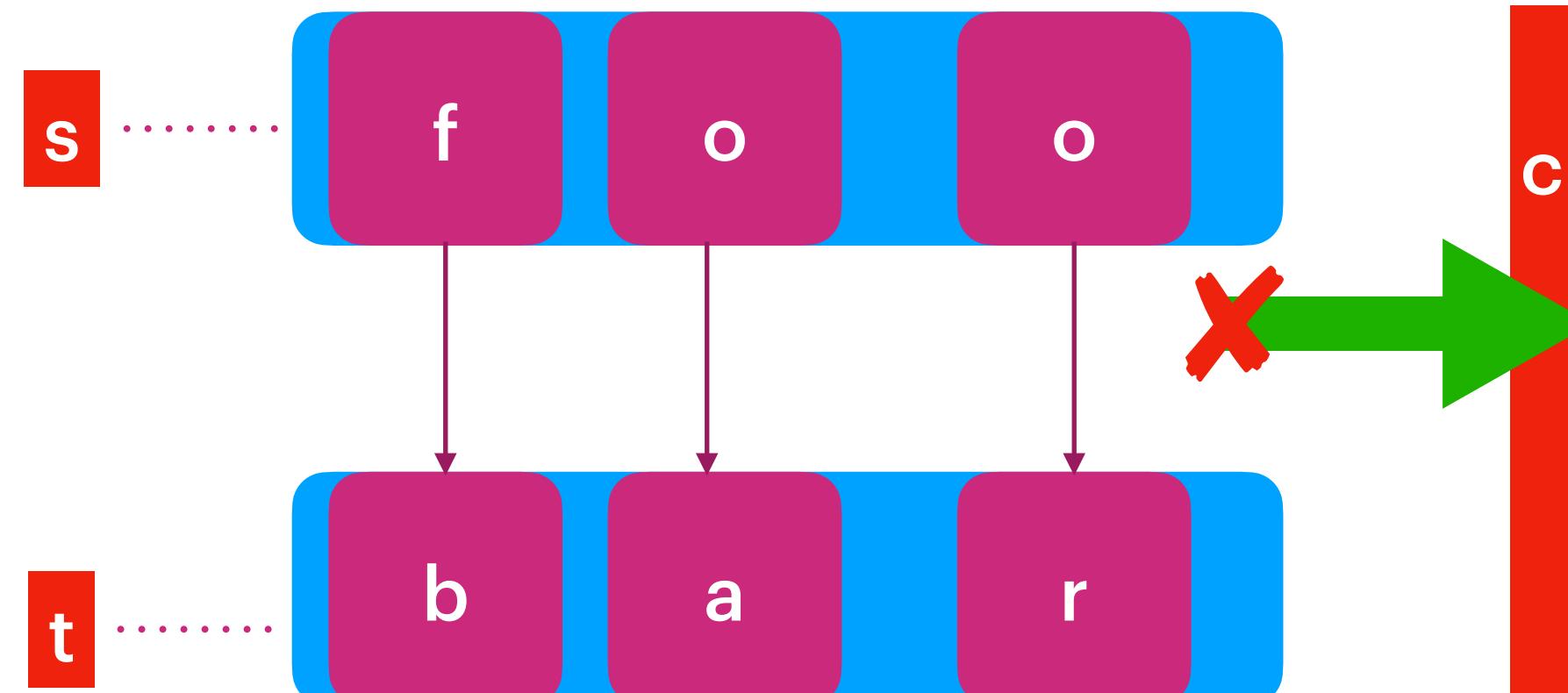
Output: false



s & t are Isomorphic, because  
Each character in 's' is uniquely mapped to 't'.

s    t  
e -> a  
g -> d  
g -> d

Advantage: if you replace each character of 's' with respective character in 't', you will get "add"



s & t are not Isomorphic, because  
character 'o' in 's' is mapped to multiple character in 't'.

s    t  
f -> b  
o -> a  
o -> r

s & t are Isomorphic

If and Only If each character in s can map to each character in t uniquely

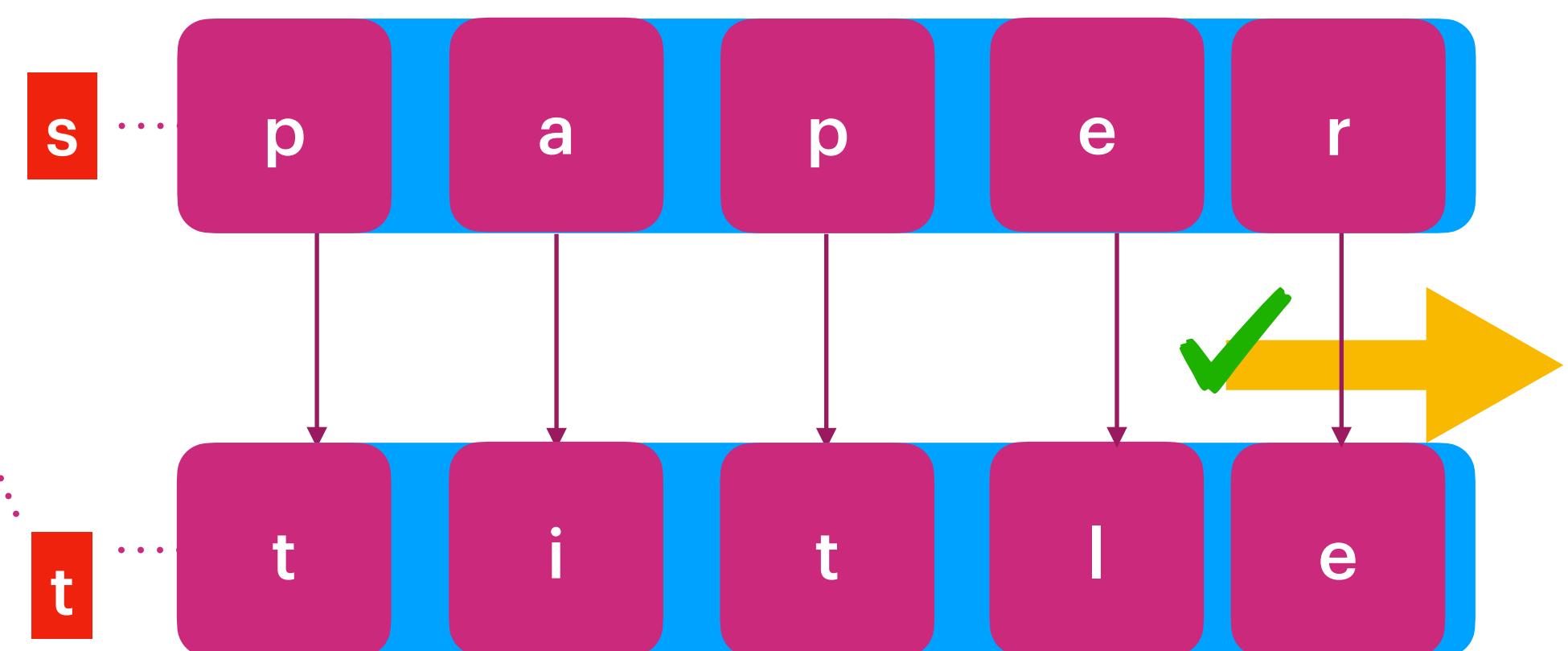
In simple single character in 't' can not be mapped to multiple character in 's' and also single character in 't' can not be mapped multiple characters in 's'.

Input: s = "egg", t = "add"

Output: true

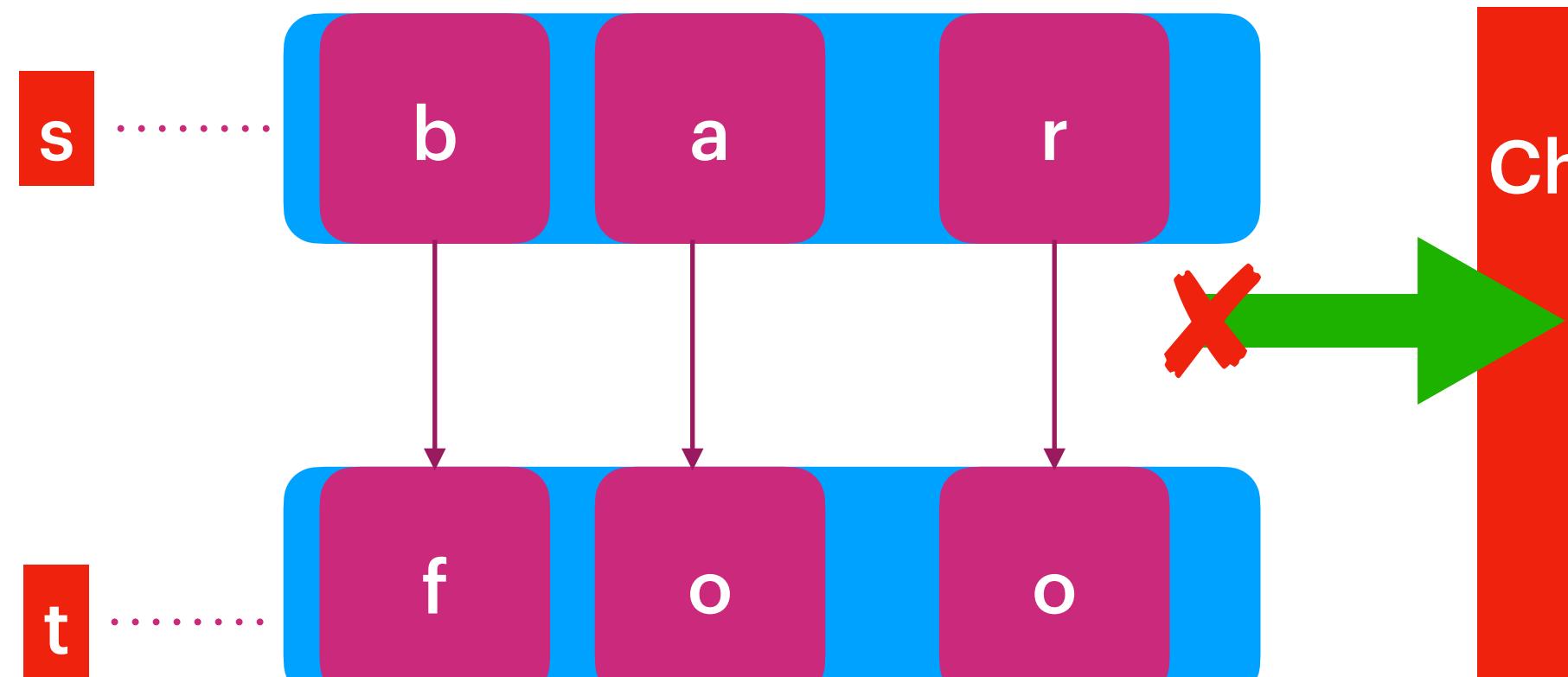
Input: s = "foo", t = "bar"

Output: false



s & t are Isomorphic, because  
Each character in 's' is uniquely mapped to 't'.

s    t  
p -> t  
a -> i  
p -> t  
e -> l  
r -> e



s & t are not Isomorphic, because  
Characters 'o' in t is mapped to multiple character in 's'.

t    s  
f -> b  
o -> a  
o -> r

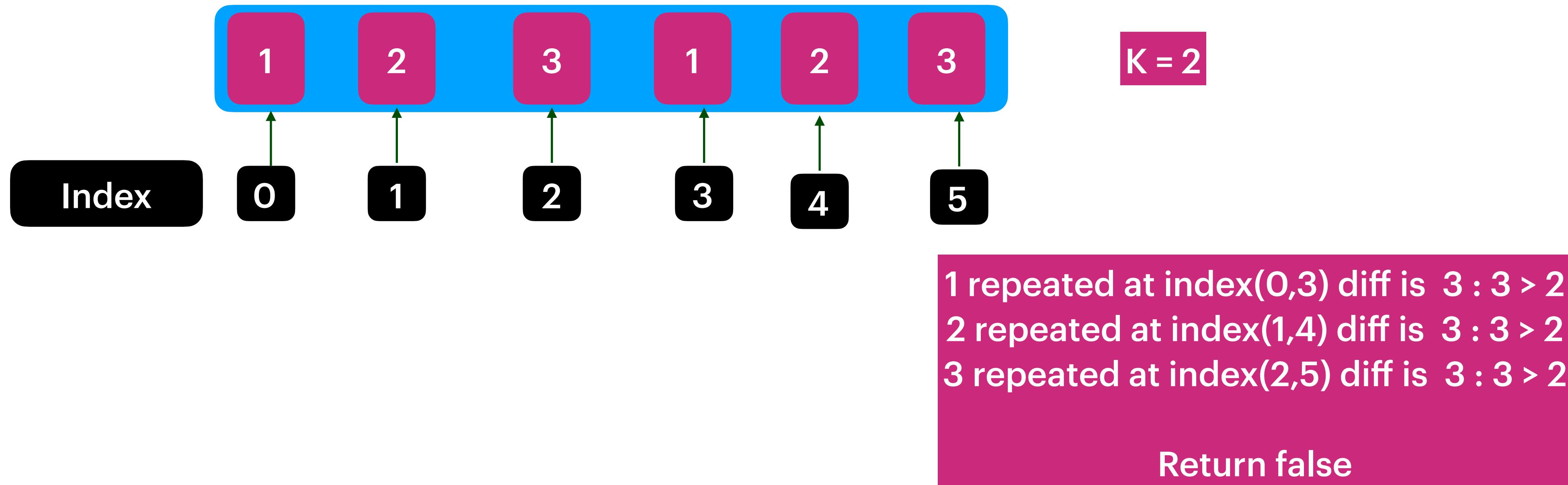
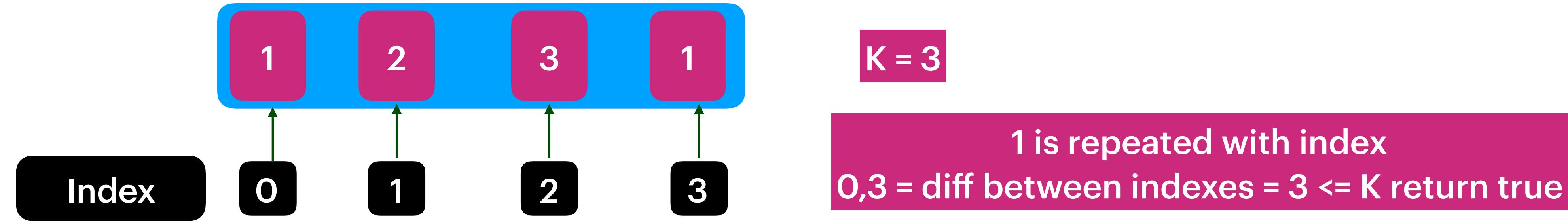
## Contains Duplicate II

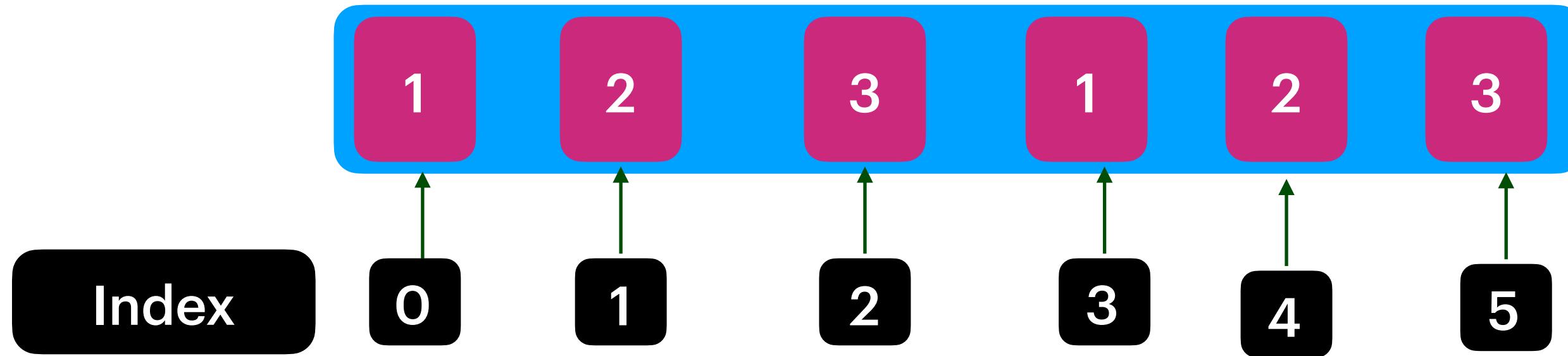
Given an integer array **nums** and an integer **k**, return true if there are two distinct indices **i** and **j** in the array such that **nums[i] == nums[j]** and  $\text{abs}(i - j) \leq k$ .

**Input:** **nums** = [1,2,3,1], **k** = 3  
**Output:** true

**Input:** **nums** = [1,0,1,1], **k** = 1  
**Output:** true

**Input:** **nums** = [1,2,3,1,2,3], **k** = 2  
**Output:** false



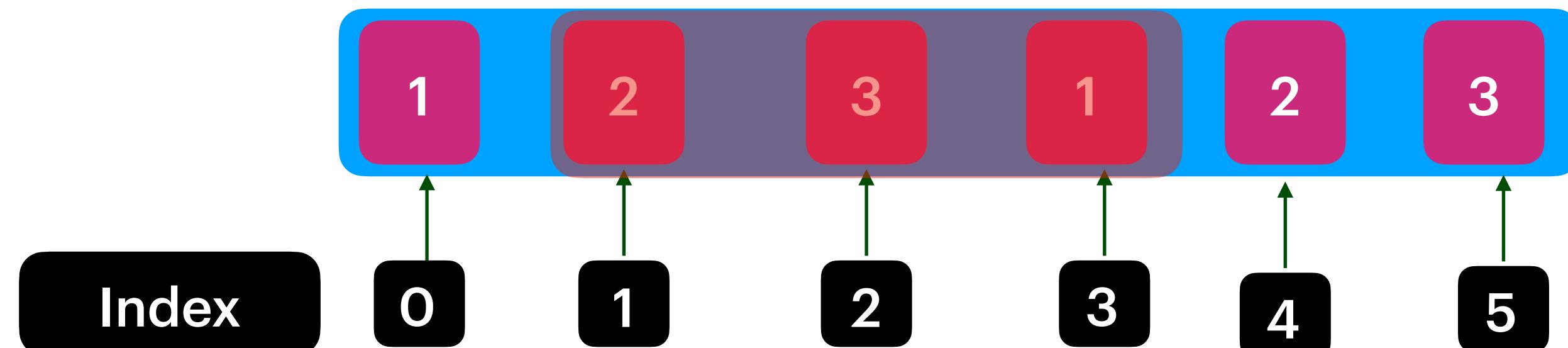
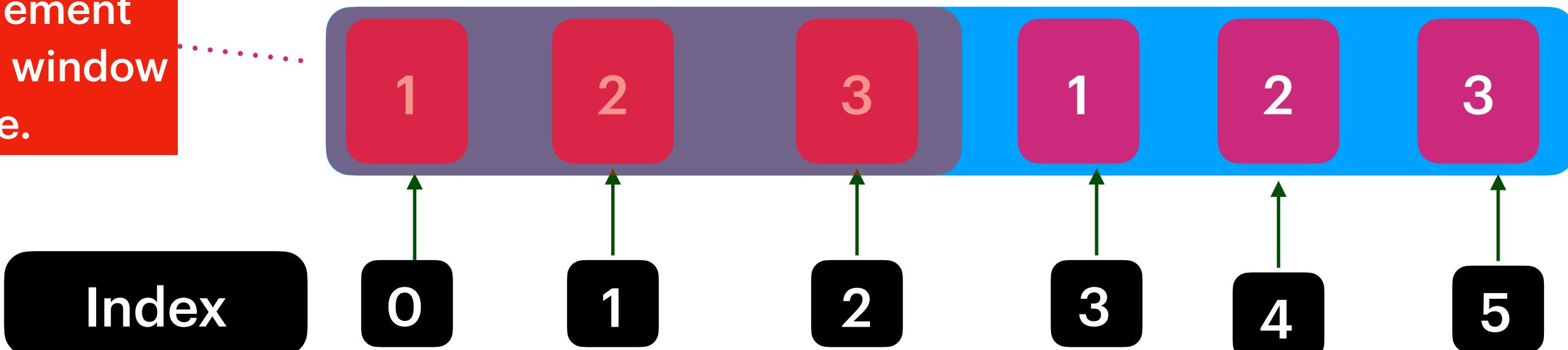


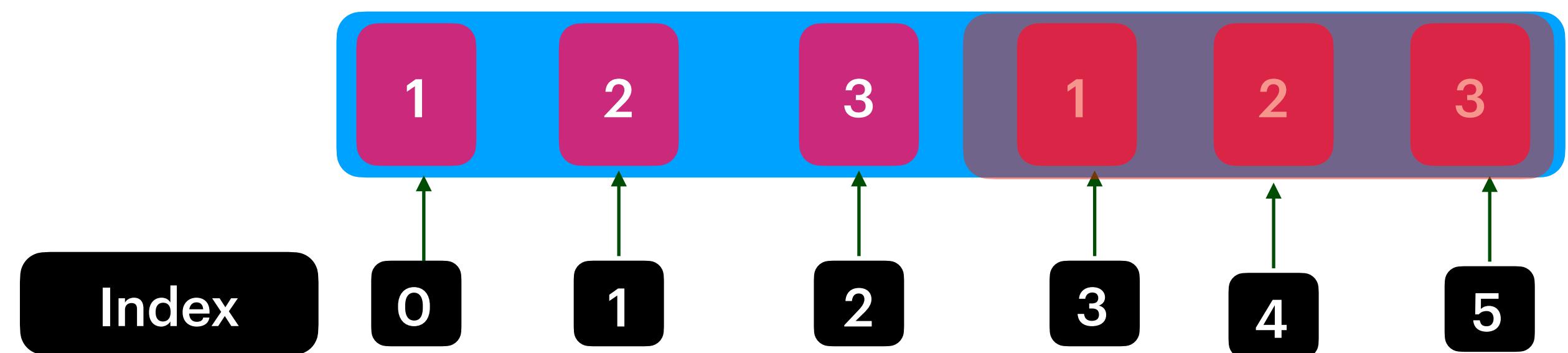
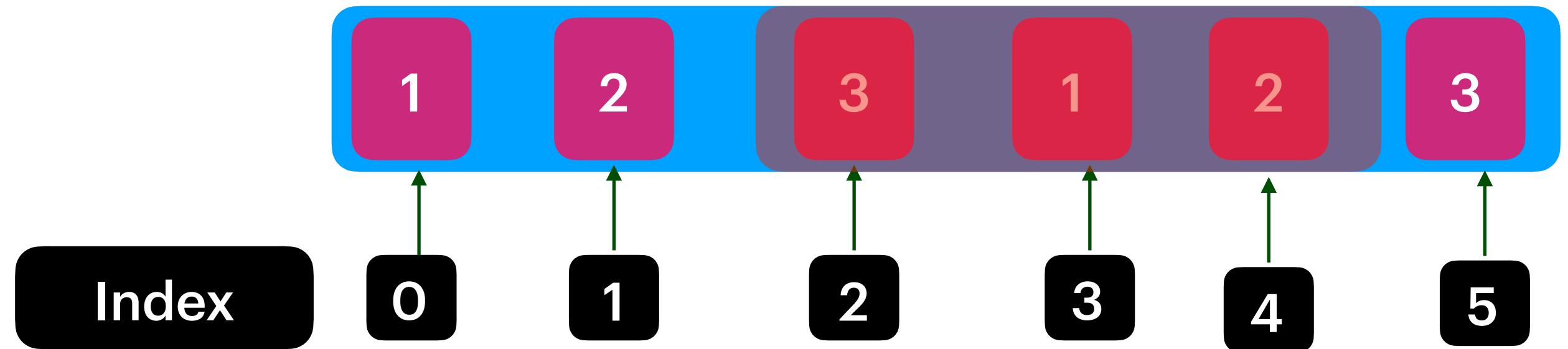
$K = 2$

1 repeated at index(0,3) diff is  $3 : 3 > 2$   
 2 repeated at index(1,4) diff is  $3 : 3 > 2$   
 3 repeated at index(2,5) diff is  $3 : 3 > 2$

Return false

Probability of success rate for a given element is within a window Size.





## Group Anagrams

Given an array of strings `strs`, group the anagrams together. You can return the answer in any order.

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once. All the characters are in lower case

**Input:** `strs = ["eat","tea","tan","ate","nat","bat"]`  
**Output:** `[["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]`

**Input:** `strs = [""]`  
**Output:** `[[""]]`

**Input:** `strs = ["a"]`  
**Output:** `[["a"]]`

```
strs = ["are", "bat", "ear", "code", "tab", "era"]
```

### Java

```
ans = {"aer": ["are", "ear", "era"],  
       "abt": ["bat", "tab"],  
       "ecdo": ["code"]}
```

**Key in all the Anagram words have the same characters !!! Simply sort the words , have a key with sorted characters.**

Ex : => "eat","ate","tea" here when we sort each word you get the same key : "aet"

**Input:**

String strings= ["eat","tea","tan","ate","nat","bat"]

"aet" = ["eat","ate","tea"]

"ant" = ["tan","nat"]

"bat" = ["bat"]

Apply hashing on sorted Key.  
Time Complexity :  $O(nk\log(k))$   
Space Complexity :  $O(n)$

## Do we really need Sorting ?

Lets apply math here , as we know that characters are in lower case and the max key we can make is 26 characters.

Have Character[] of size 26.

In each step before processing input String fill the Character[] with null values.

Gets the index of each character , make sure that current character placed into appropriate index.

```
int index = s.charAt(i) - 'a';
```

Just traverses the Character array form a key where index value is not null.

Time Complexity :  $O(nk)$

Space Complexity :  $O(n)$

String strings= ["abc","cba","bac","dad","add"]

[ [ "abc" , "cba" , "bac"], [ "dad", "add" ] ]

"cab"

$$'c' - 'a' = 98 - 96 = 2$$

$$'a' - 'a' = 96 - 96 = 0$$

$$'b' - 'a' = 97 - 96 = 1$$

{ a,b,c,null....null}

"abc"

$$'a' - 'a' = 96 - 96 = 0$$

$$'b' - 'a' = 97 - 96 = 1$$

$$'c' - 'a' = 98 - 96 = 2$$

{ a,b,c,null....null}

"dad"

$$'d' - 'a' = 99 - 96 = 3$$

$$'a' - 'a' = 96 - 96 = 0$$

$$'d' - 'a' = 99 - 96 = 3$$

{ a, null , null ,d, ....null}

"add"

$$'a' - 'a' = 96 - 96 = 0$$

$$'d' - 'a' = 99 - 96 = 3$$

$$'d' - 'a' = 99 - 96 = 3$$

{ a, null , null ,d, ....null}

Words “cab” & “abc” formed a same array with {a, b,c, Null.... Null}

Just travers the array form a key where index value is not null.

So key is “abc” map the values .

Done => abc = {"cab","abc"}

Words “dad” & “add” formed a same array with {a, null, null, d.... Null}

Just travers the array form a key where index value is not null.

So key is “ad” map the values .

Done => ad = {"dad","add"}

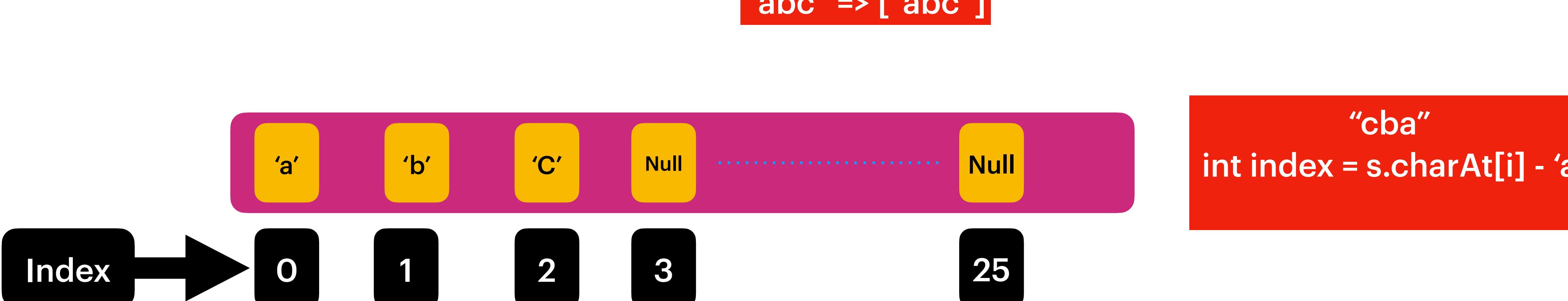
```
String strings= ["abc","cba","bac","dad","add"]
```

```
[ [ "abc" , "cba" , "bac"], [ "dad", "add" ] ]
```

In each step before processing  
input String fill the Character[] with null values.



“abc” => [“abc”]



“abc” => [“abc”, “cba”]

String strings= ["abc","cba","bac","dad","add"]

[ [ "abc" , "cba" , "bac"], [ "dad" , "add" ] ]

In each step before processing  
input String fill the Character[] with null values.



"bac"  
int index = s.charAt[i] - 'a'

"abc" => [ "abc" , "cba" , "bac" ]

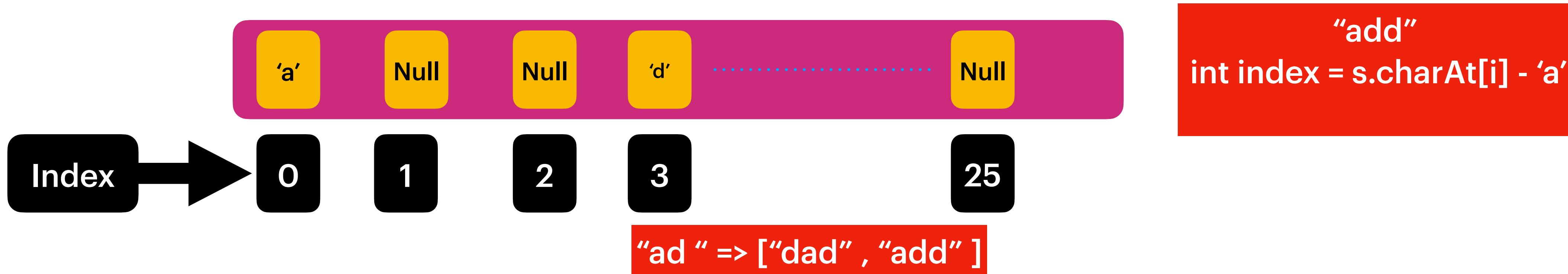


"dad"  
int index = s.charAt[i] - 'a'

"ad " => [ "dad" ]

```
String strings= ["abc","cba","bac","dad","add"]  
[ [ "abc" , "cba" , "bac"], ["dad","add"]]
```

In each step before processing  
input String fill the Character[] with null values.



## Group Shifted Strings

can shift a string by shifting each of its letters to its successive letter.

For example, "abc" can be shifted to be "bcd".

We can keep shifting the string to form a sequence.

For example, we can keep shifting "abc" to form the sequence: "abc" -> "bcd" -> ... -> "xyz". Given an array of strings strings, group all strings[i] that belong to the same shifting sequence. You may return the answer in any order.

Constraints:

1 <= strings.length <= 200

1 <= strings[i].length <= 50

strings [i] consists of lowercase English letters.

Input: strings = ["abc","bcd","acef","xyz","az","ba","a","z"]

Output: [["acef"], ["a", "z"], ["abc", "bcd", "xyz"], ["az", "ba"]]

Input: strings = ["a"]

Output: [[["a"]]]

## Lets look at Group Shifted Strings

a -> b -> c -> .....-> z -> a -> b

ab -> bc -> cd ->....->yz->za->ab

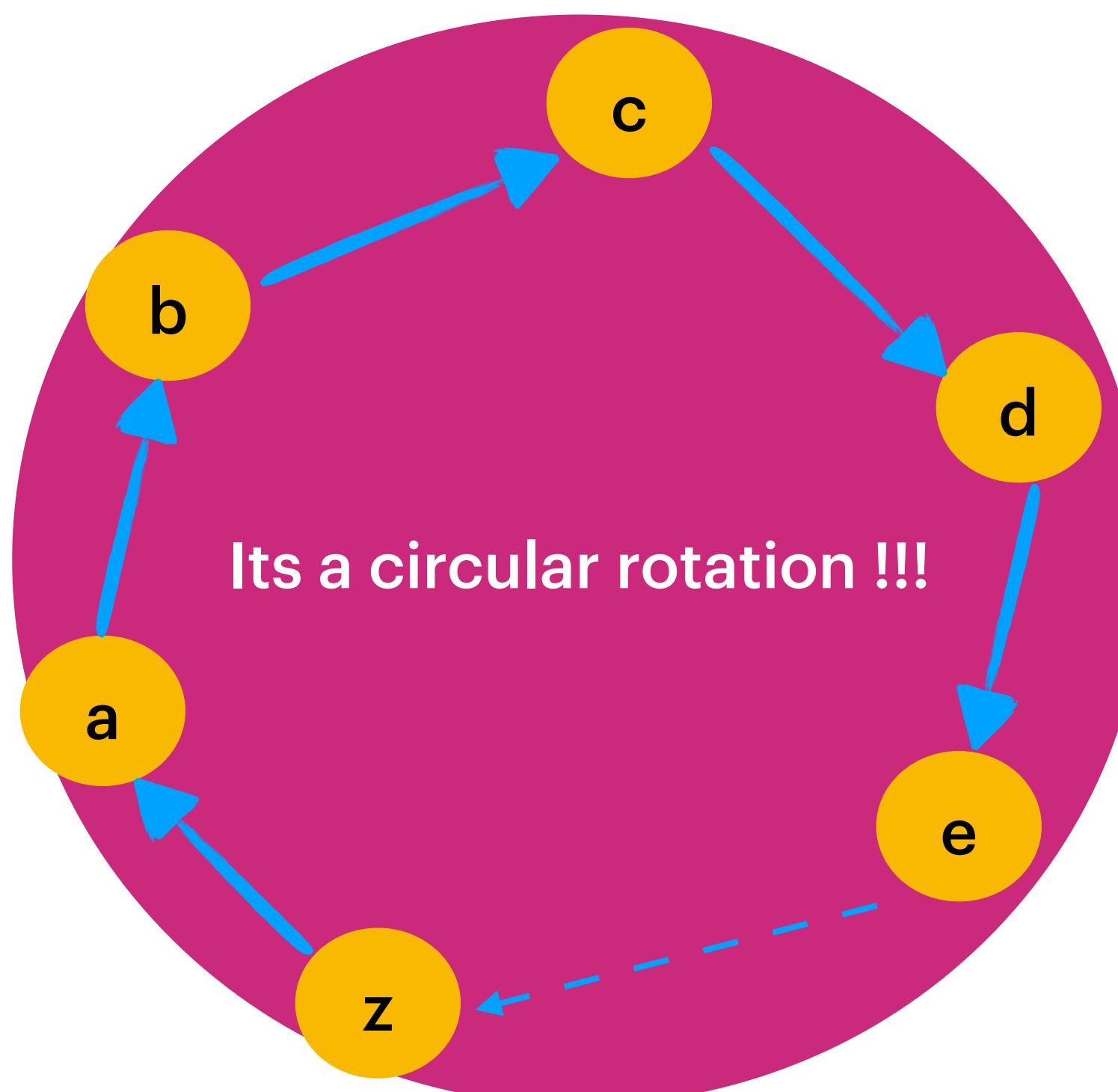
abc -> bcd -> cde -> ....> xyz -> yza ->zab -> abc

az -> ba

We find distance of each character

It should be starting character of the string because rotation start from here !!  
This enables us to give GroupKey.

$( s.charAt(i) - s.charAt(0) + 26 ) \% 26$



"abc", "bcd", "xyz"

Distance between characters (c->b->a) is equal

"az", "ba"

Distance between characters (a -> b), (z->a) is equal

Let's Apply math here to form a Group Key

**Principle = (s.charAt(i) - startingChar + 26) % 26**

String s = "abc"  
startingChar = 'a'  
 $(a - a + 26) \% 26 = 0$   
 $(b - a + 26) \% 26 = (97-96+26)\% 26 = 1$   
 $(c - a + 26) \% 26 = (98-96+26)\% 26 = 2$   
  
Finally "bcd" key = "0|1|2|"

String s = "cde"  
startingChar = 'c'  
 $(c - c + 26) \% 26 = 0$   
 $(d - c + 26) \% 26 = (99-98+26)\% 26 = 1$   
 $(e - c + 26) \% 26 = (100-98+26)\% 26 = 2$   
  
Finally "cde" key = "0|1|2|"

String s = "bcd"  
startingChar = 'b'  
 $(b - b + 26) \% 26 = 0$   
 $(c - b + 26) \% 26 = (98-97+26)\% 26 = 1$   
 $(d - b + 26) \% 26 = (99-97+26)\% 26 = 2$   
  
Finally "bcd" key = "0|1|2|"

"abc", "bcd", "xyz"

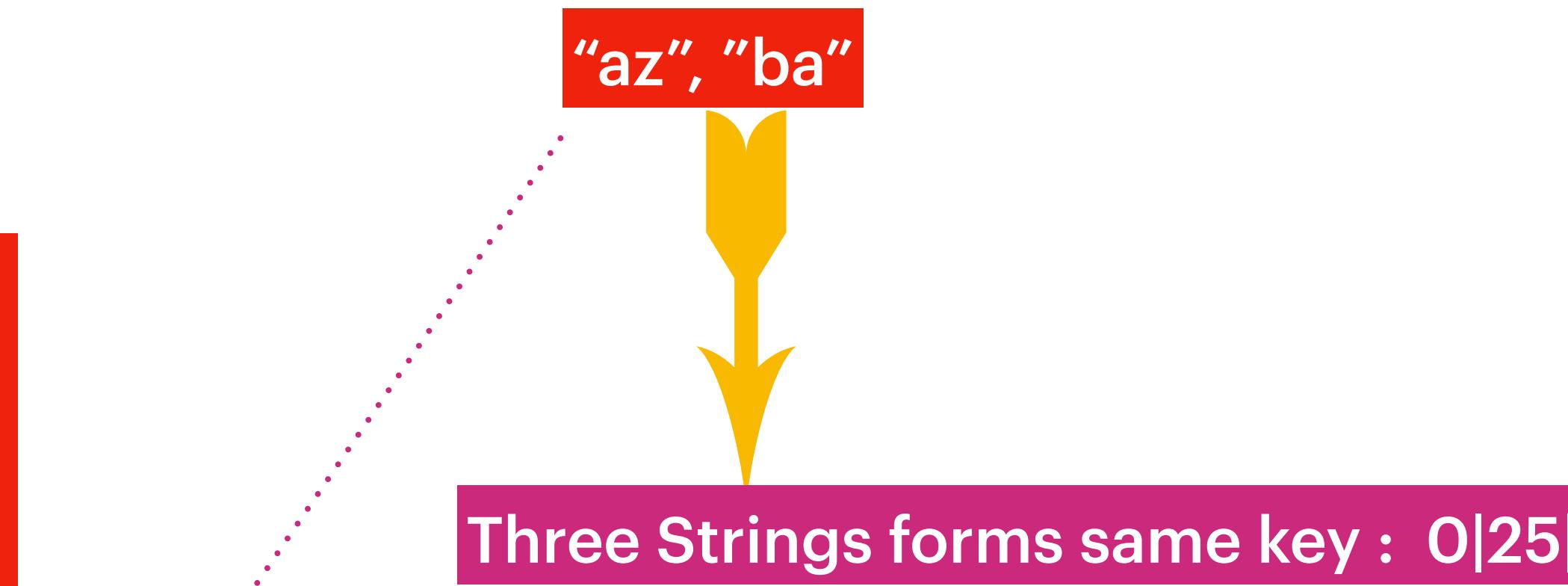


Three Strings forms same key : 0|1|2|

Principle = (s.charAt(i) - startingChar + 26) % 26

String s = "az"  
startingChar = 'a'  
 $(a - a + 26) \% 26 = 0$   
 $(z - a + 26) \% 26 = (121-96+26)\% 26$   
 $= (25 + 26)\%26 = 25$   
key = 0|25|

String s = "ba"  
startingChar = 'b'  
 $(b - b + 26) \% 26 = 0$   
 $(a - b + 26) \% 26 = (96-97+26)\% 26$   
 $= (25 \% 26) = 25$   
Key = 0|25|



# Longest Substring Without Repeating Characters

Given a string **s**, find the length of the **longest substring** without repeating characters.

**Constraints:**

$0 \leq s.length \leq 5 * 10^4$

**s** consists of English letters, digits, symbols and spaces.

**Example 1:**

**Input:** s = "abcabcbb"

**Output:** 3

**Explanation:** The answer is "abc", with the length of 3.

**Example 2:**

**Input:** s = "srinu nampalli"

**Output:** 6

**Explanation:** The answer is "srinu ", with the length of 1.

**Example 3:**

**Input:** s = "pwwkew"

**Output:** 3

**Explanation:** The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

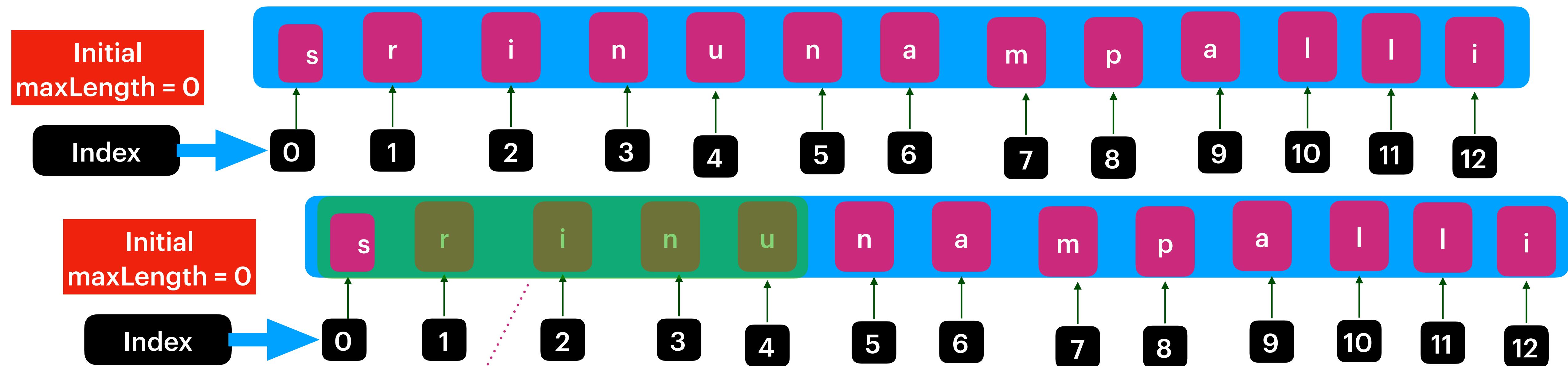
**Example 4:**

**Input:** s = ""

**Output:** 0

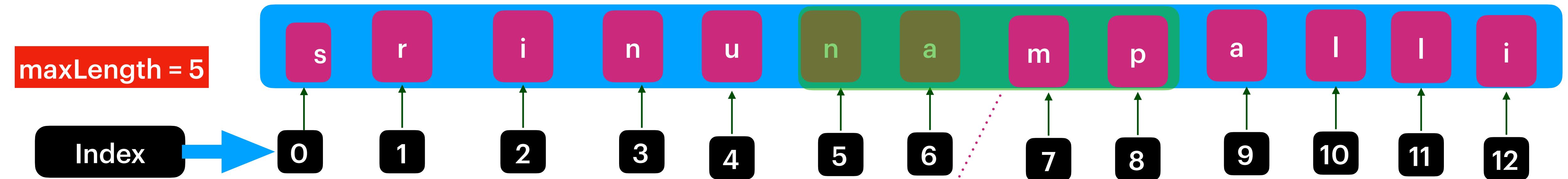
## Abstract Solution with HashSet :

Take the character set, store the data into set till the character is not repeated.  
Update the maxLength and reset the size, repeat the process from next character onwards .

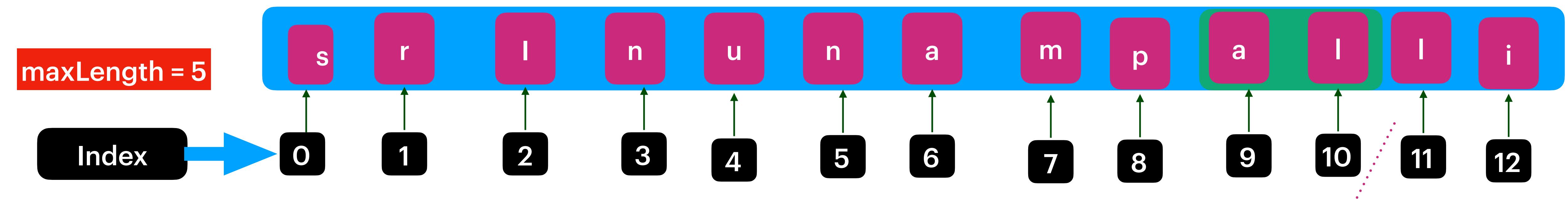


Store the data into `Set<Character>` till the character is not repeated :  
set Size =5  
current maxLength = 0 so update maxLength = 5  
empty the Set.

## Abstract Solution with HashSet :



Move on to next Character, Store the data into Set<Character> till the character is not repeated :  
set Size = 4  
current maxLength = 5 so no update maxLength = 5  
empty the Set.

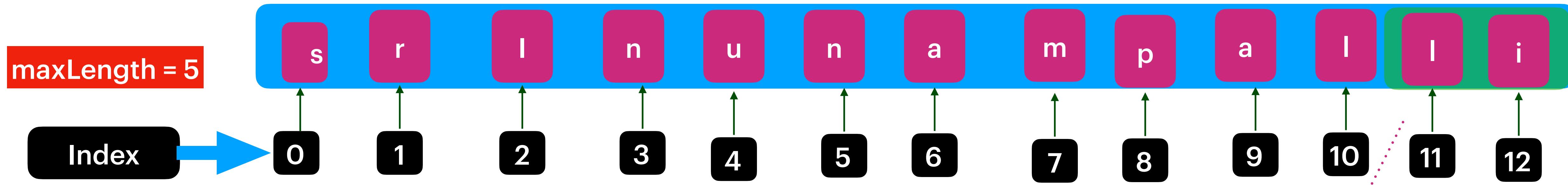


Move on to next Character, Store the data into Set<Character> till the character is not repeated :  
set Size = 2  
current maxLength = 5 so no update on maxLength = 5  
empty the Set.

Reached end of the array, Return Max Length 6 :

Is our algorithm is perfect ?

## Abstract Solution with HashSet :



Move on to next Character, Store the data into Set<Character> till the character is not repeated :

set Size = 2

current maxLength = 5 so no update on maxLength = 5

empty the Set.

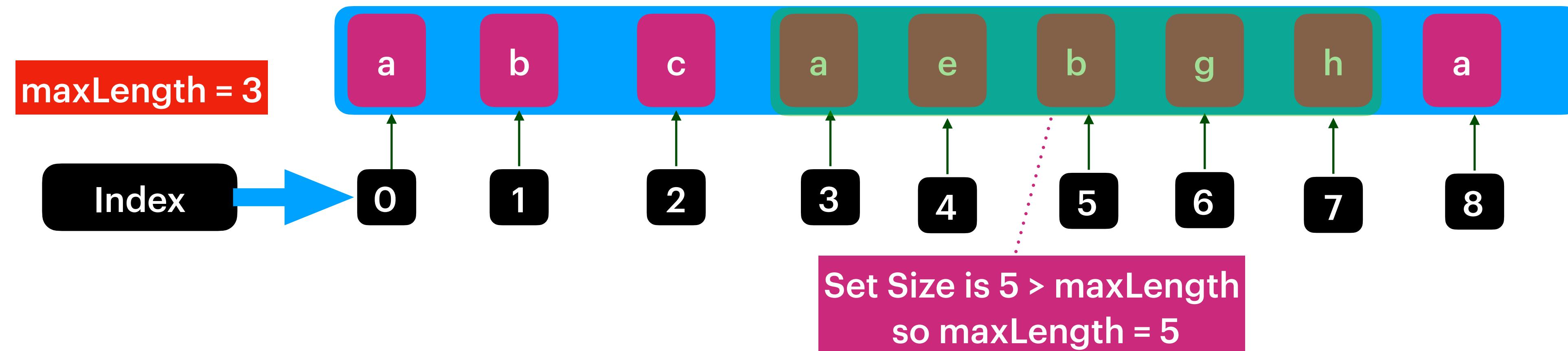
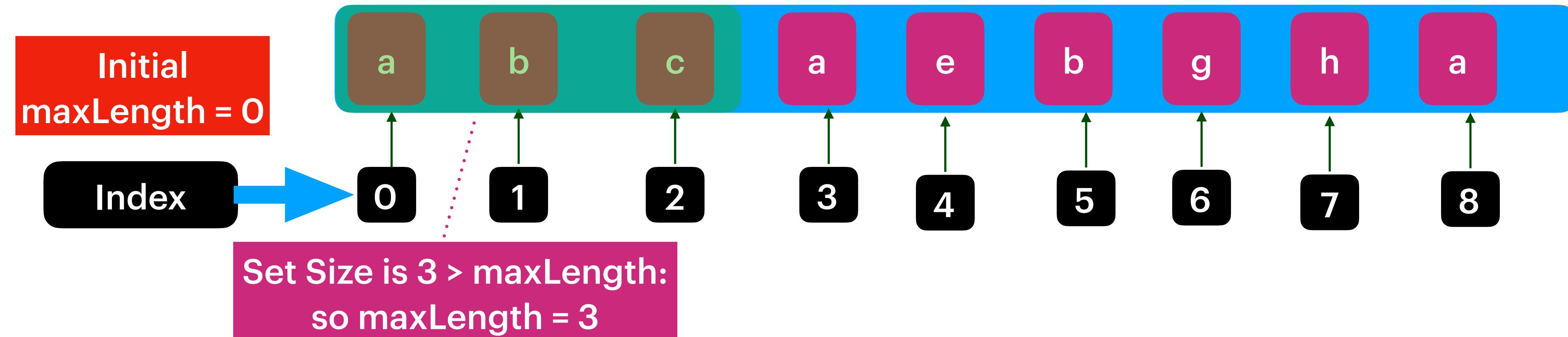
Reached end of the array, Return Max Length 5 :

Is our algorithm is perfect ?

## Abstract Solution with HashSet :

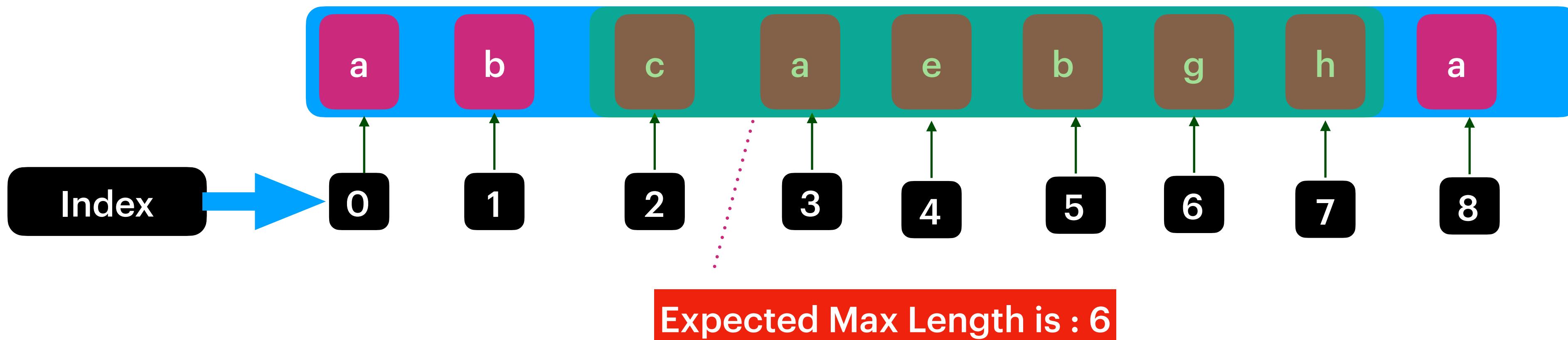
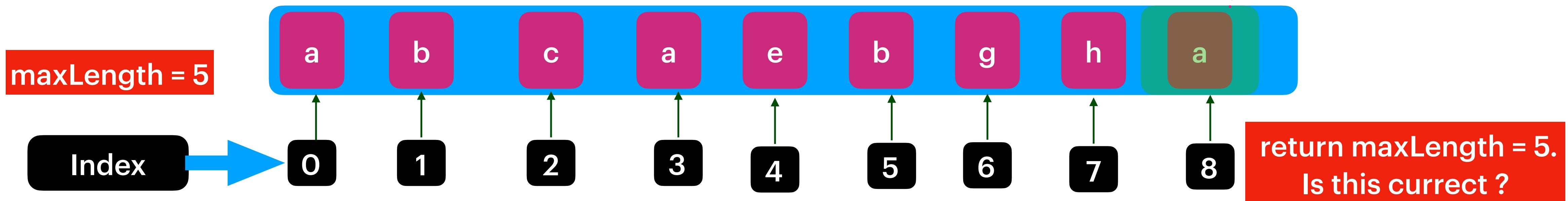
Is our algorithm is perfect ?

What if our the SubString length is higher if we consider character before the next repeated value !!!!



## Abstract Solution with HashSet :

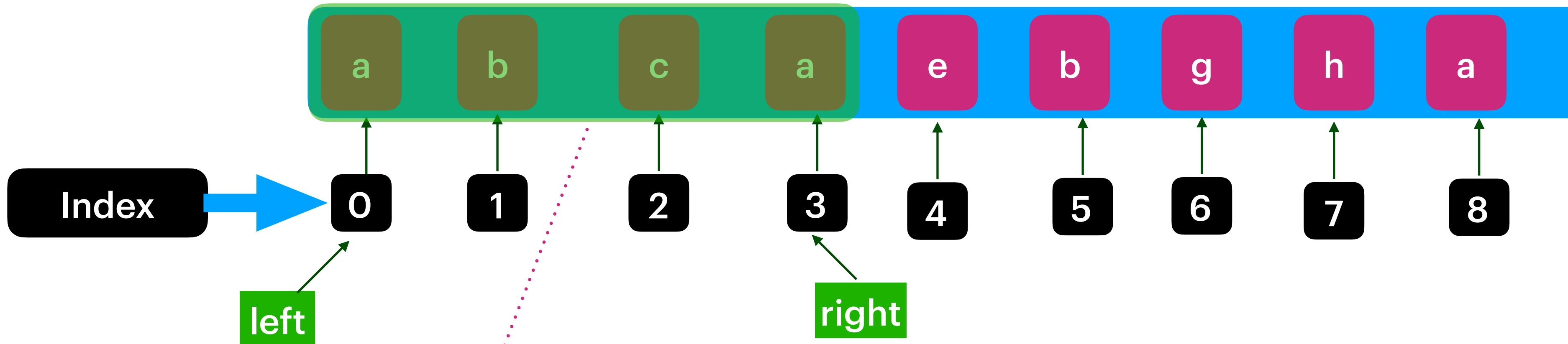
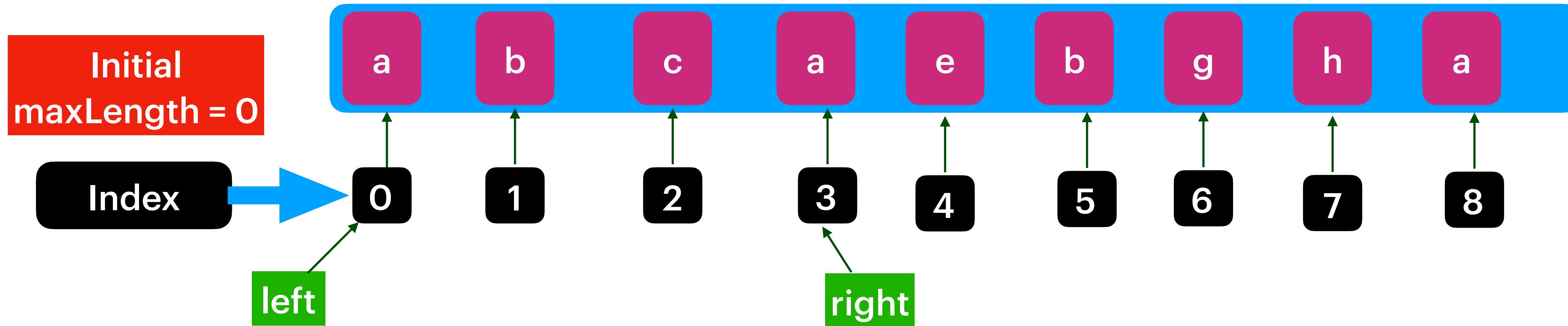
Set Size is 1:  $< \text{maxLength}$   
so no update i.e  $\text{maxLength} = 5$



Mistake from this algorithm is we are tracking left part of the window !!!!

## Abstract Solution2 : Maintaining left and right pointer !!!

Move the right pointer till the end,  
Left pointer will be moved only when the character is repeated.  
At each window take the diff of right & left pointer.



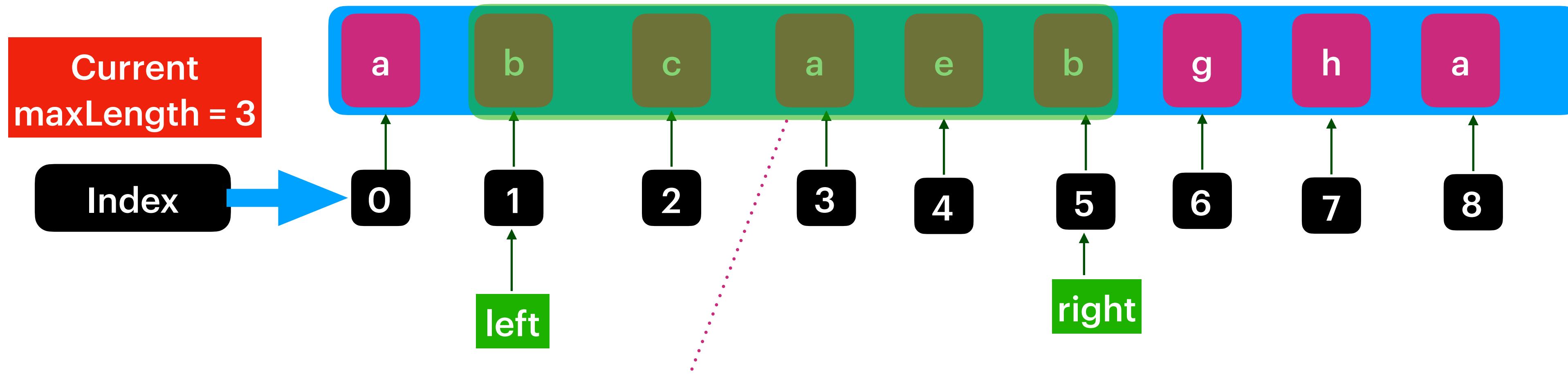
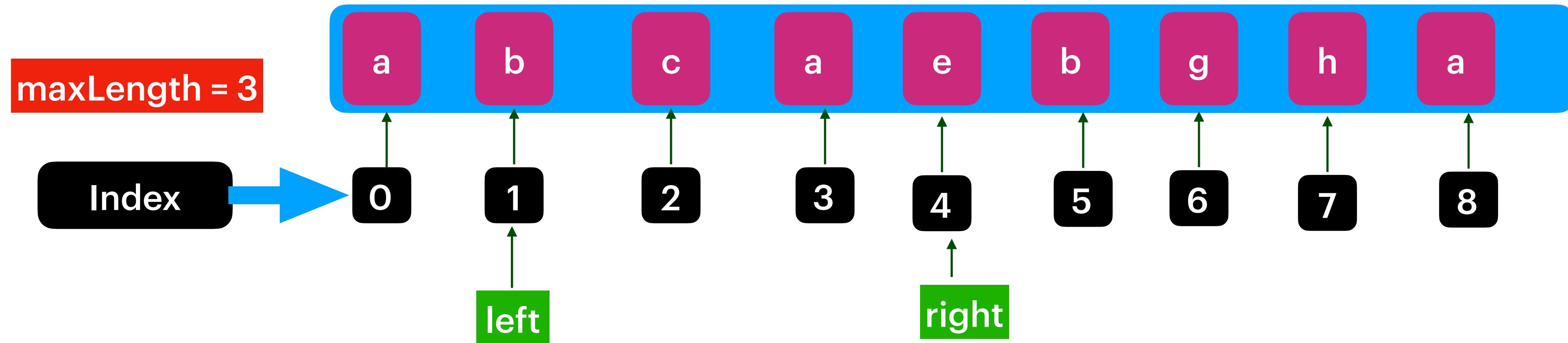
As character repeated in window update values.

$$\max = \text{Math.max}(\max, \text{right}-\text{left})$$

$$= \text{Math.max}(0, 3-0) = 3$$

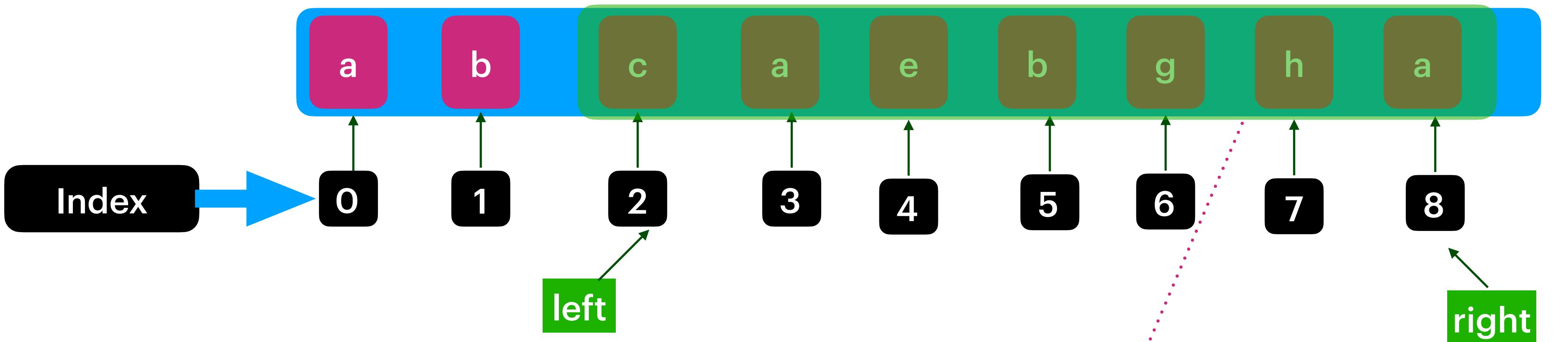
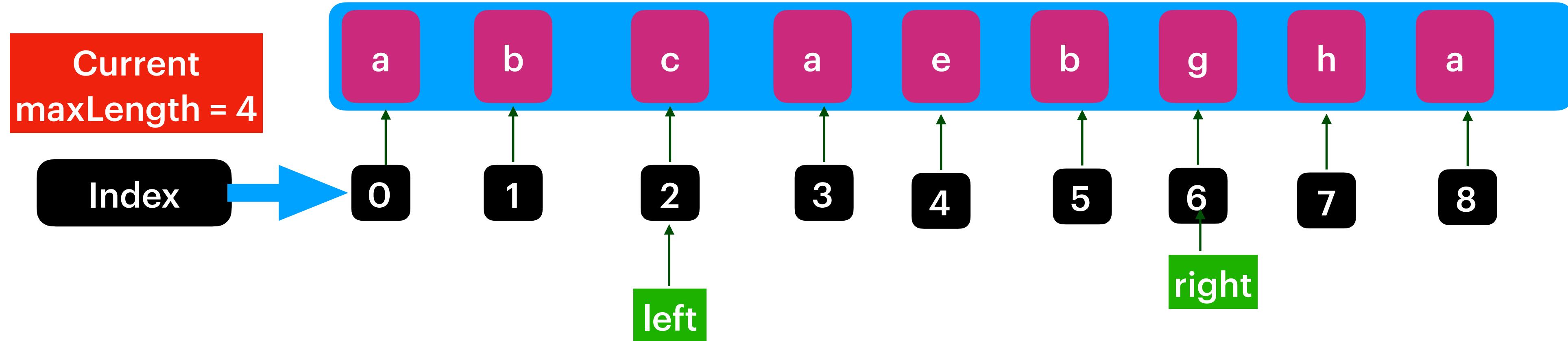
move left pointer to one index forward.

## Abstract Solution2 : Maintaining left and right pointer !!!



As character repeated in window update values.  
 $\max = \text{Math.max}(\max, \text{right-left})$   
 $= \text{Math.max}(3, 5-1) = 4$   
move left one index forward.

## Abstract Solution2 : Maintaining left and right pointer !!!



As character repeated in window update values.

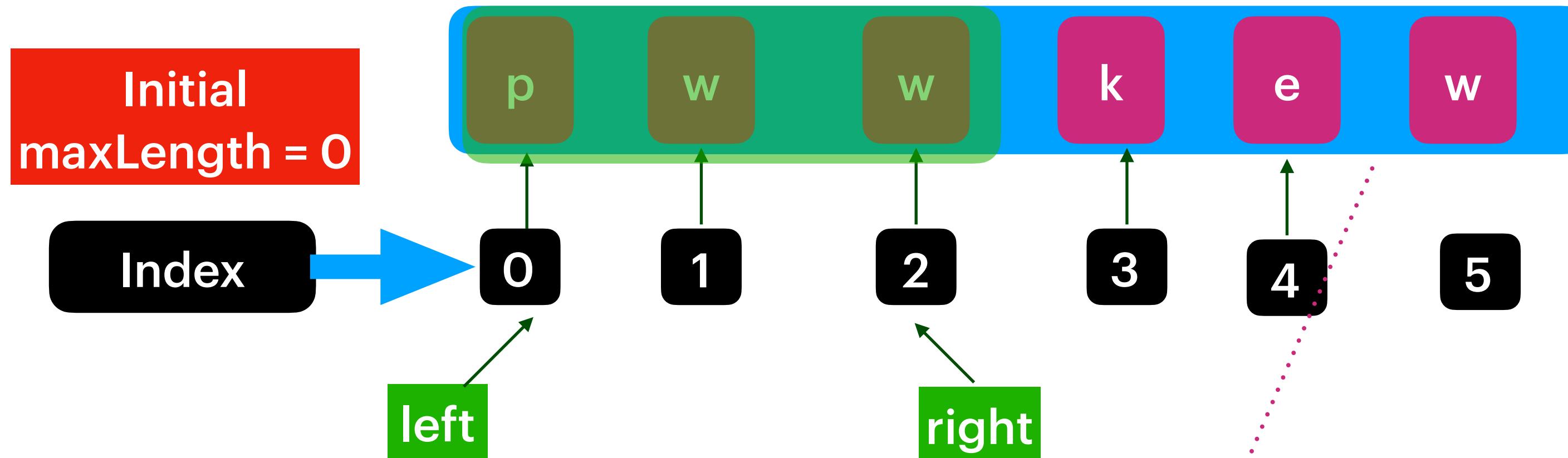
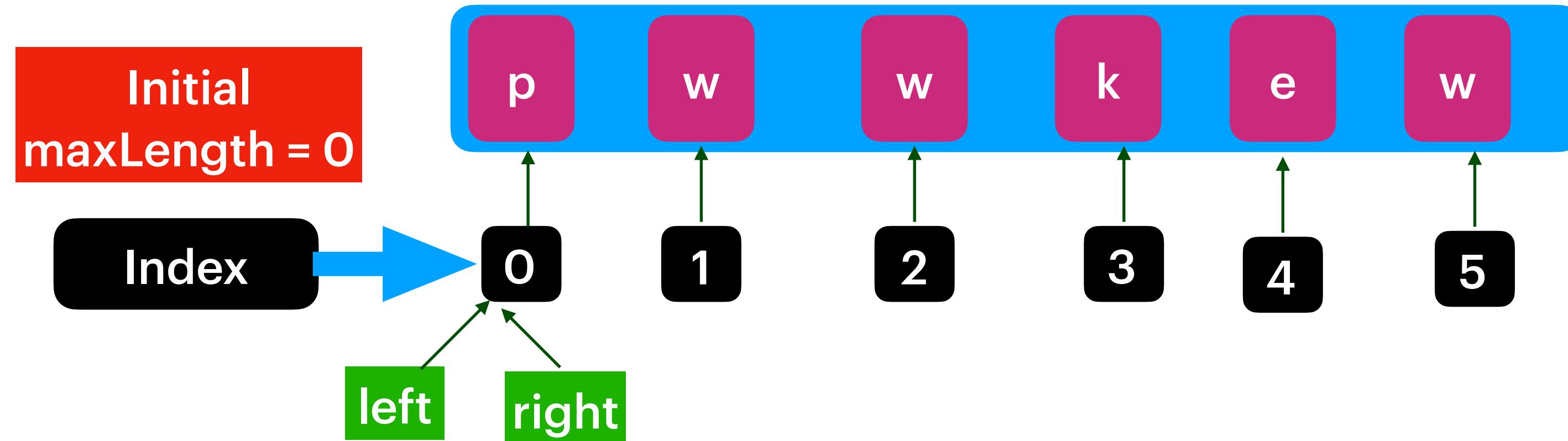
$$\max = \text{Math.max}(\max, \text{right-left})$$

$$= \text{Math.max}(4, 8-2) = 6$$

move left one index forward.

Is this Solution correct ? What if left side of window characters are duplicate ?

## Abstract Solution2 : Maintaining left and right pointer !!!



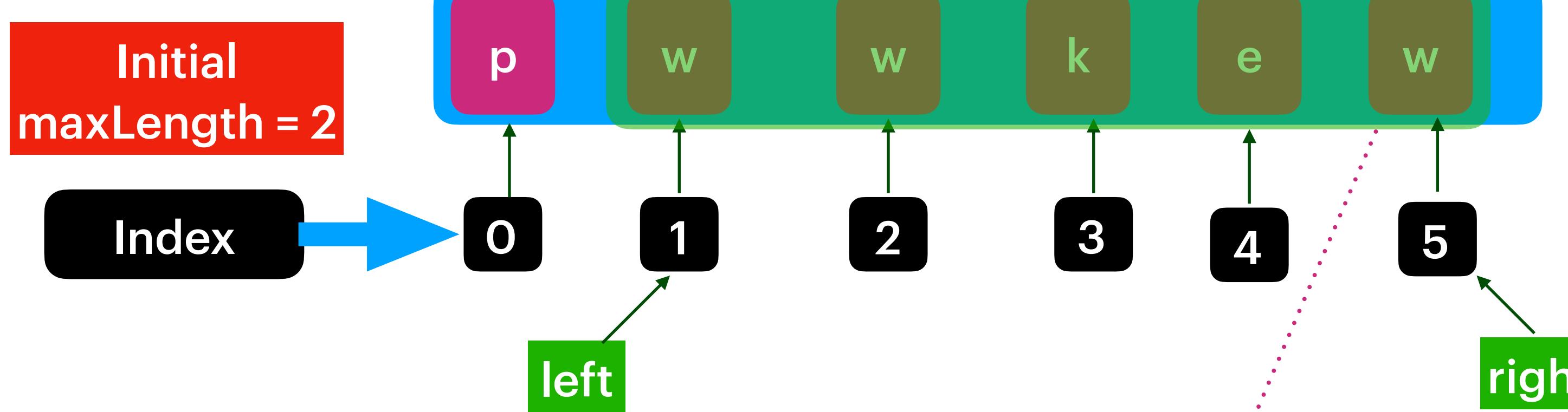
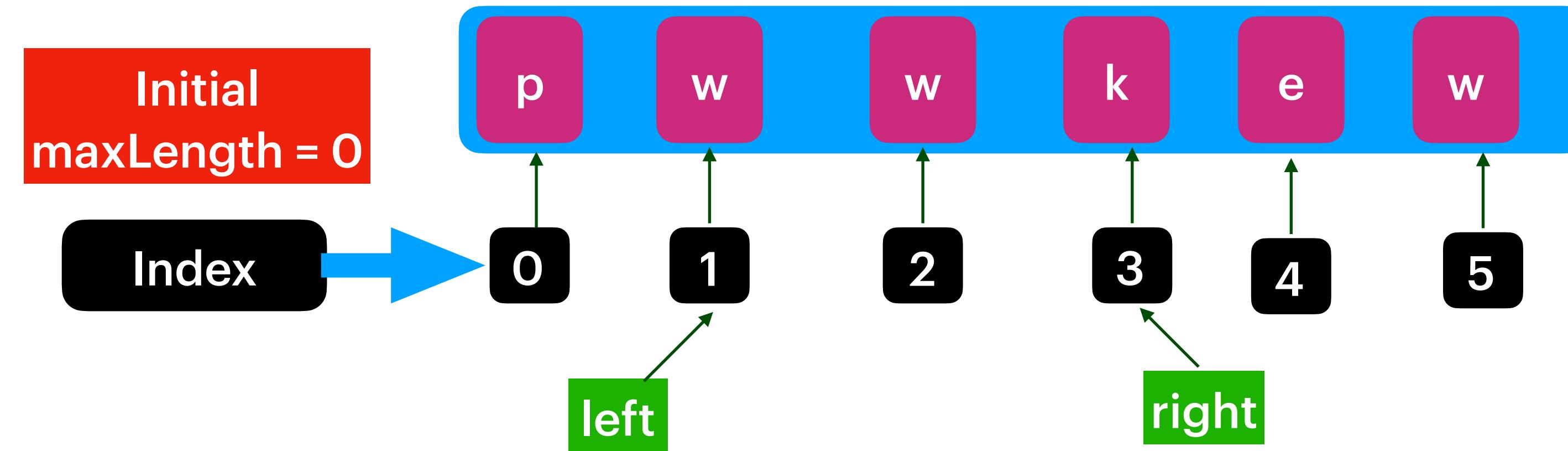
As character repeated in window update values.

$$\max = \text{Math.max}(\max, \text{right}-\text{left})$$

$$= \text{Math.max}(0, 2-0) = 2$$

move left pointer one index forward.

## Abstract Solution2 : Maintaining left and right pointer !!!

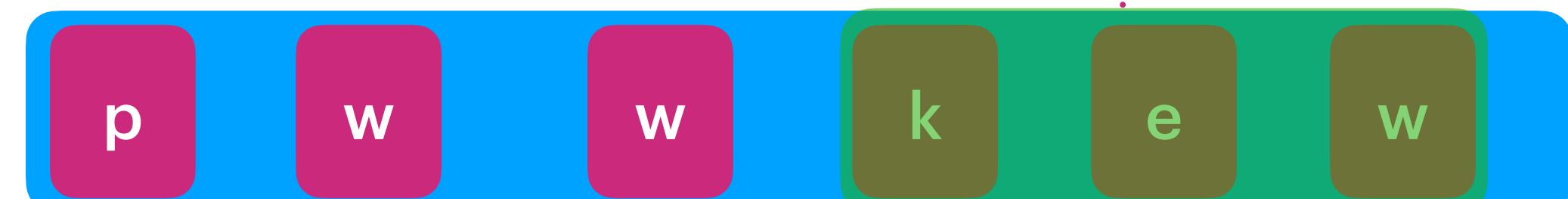


Expected Length : 3

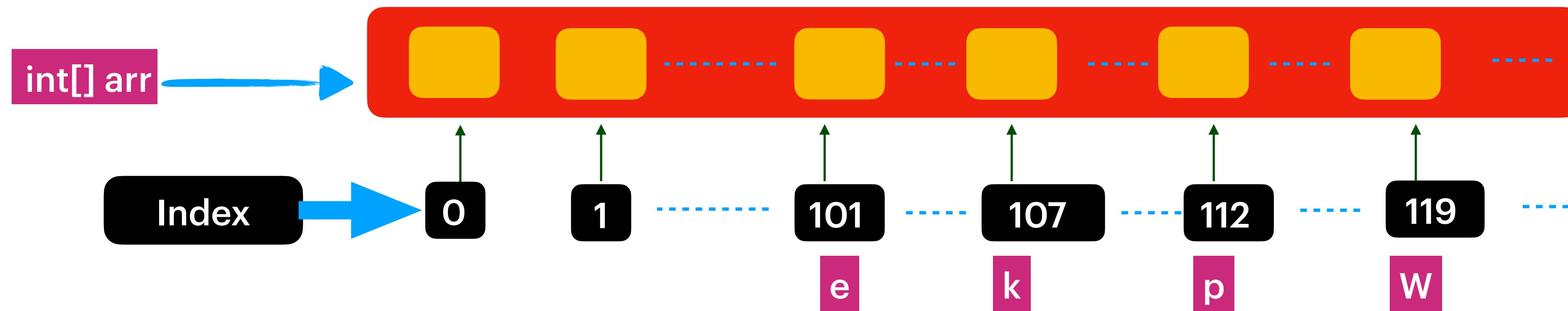
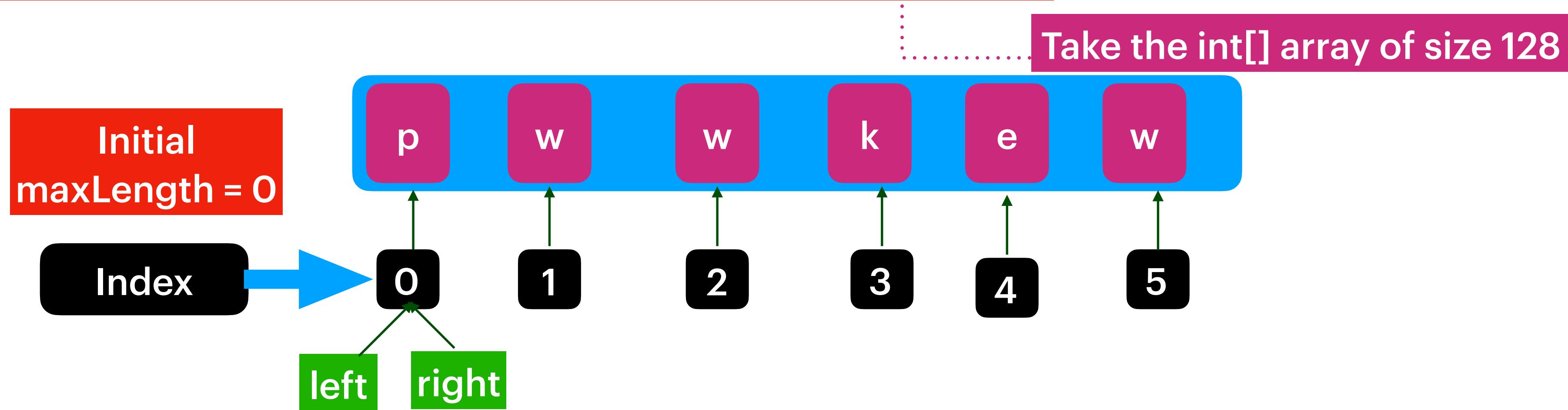
As character repeated in window update values.

$$\begin{aligned} \max &= \text{Math.max}( \max, \text{right}-\text{left}) \\ &= \text{Math.max}(2, 5-1) = 4 \end{aligned}$$

Return max Length : 4 ? Is this right Answer ?

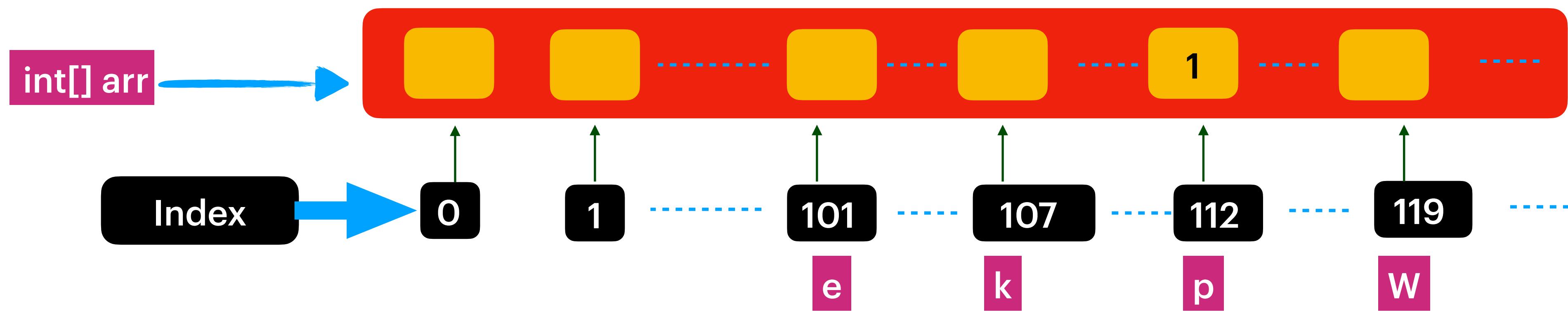
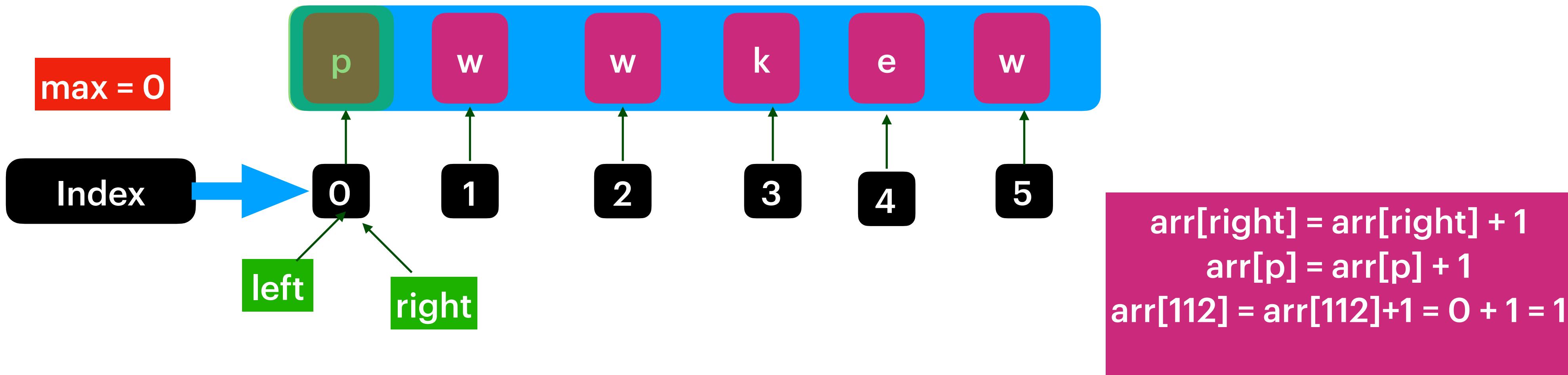


Solution is : SlidingWindow with repeated move Of left Pointer



Solution is : SlidingWindow with  
repeated move of leftPointer when there is duplicate character

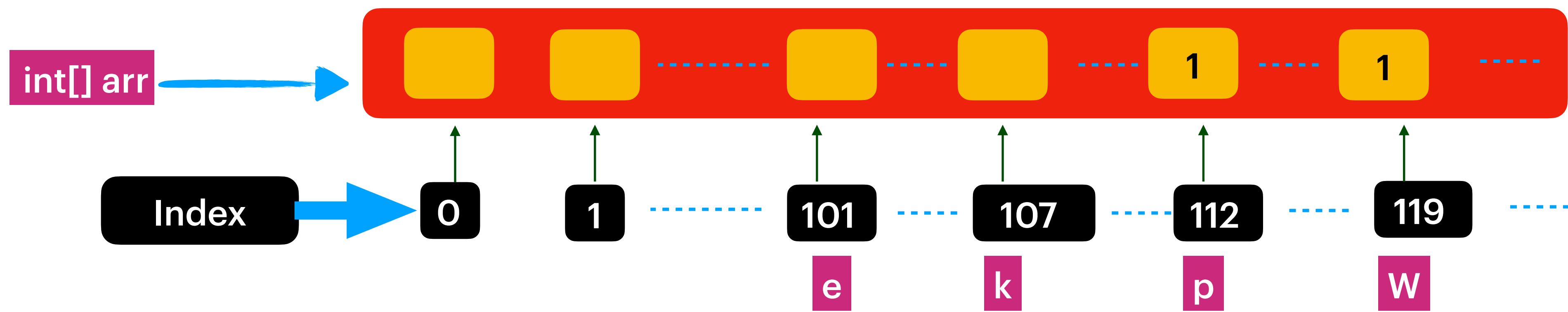
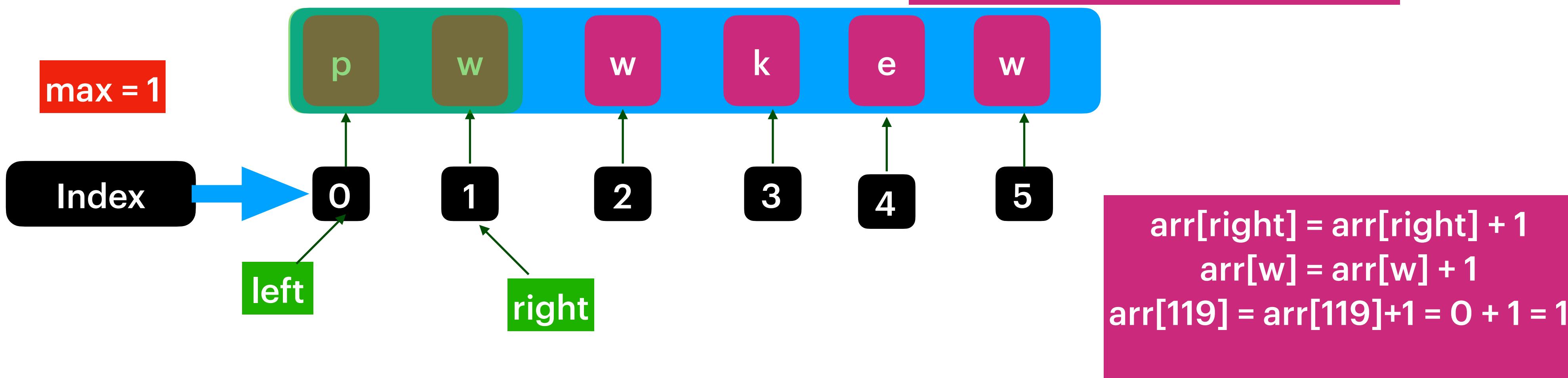
Take the int[] array of size 128  
Represents all the characters ASCII values



As arr[112] is not  $> 1$  so move right Pointer.  
 $\max = \text{Math.max}(\max, \text{right}-\text{left}+1)$   
 $\max = \text{Math.max}(0, 0-0+1) = 1$

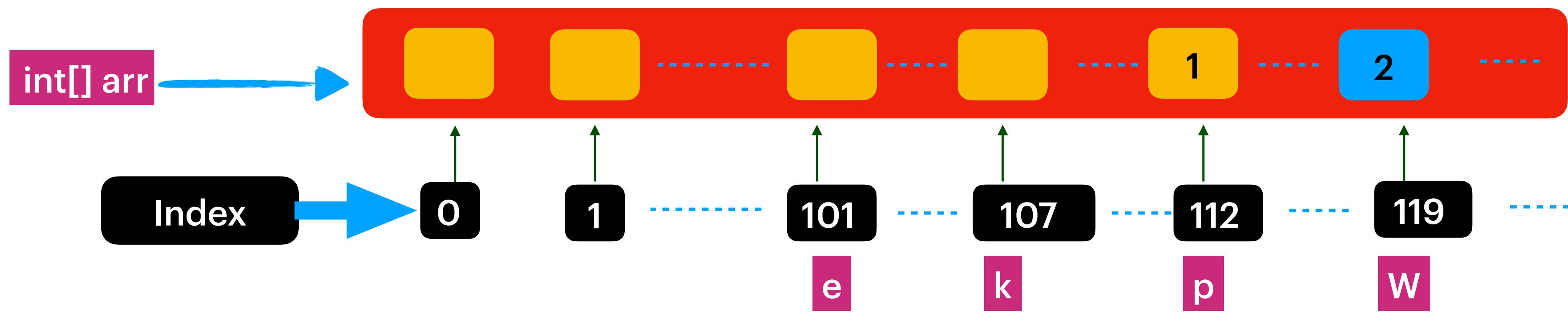
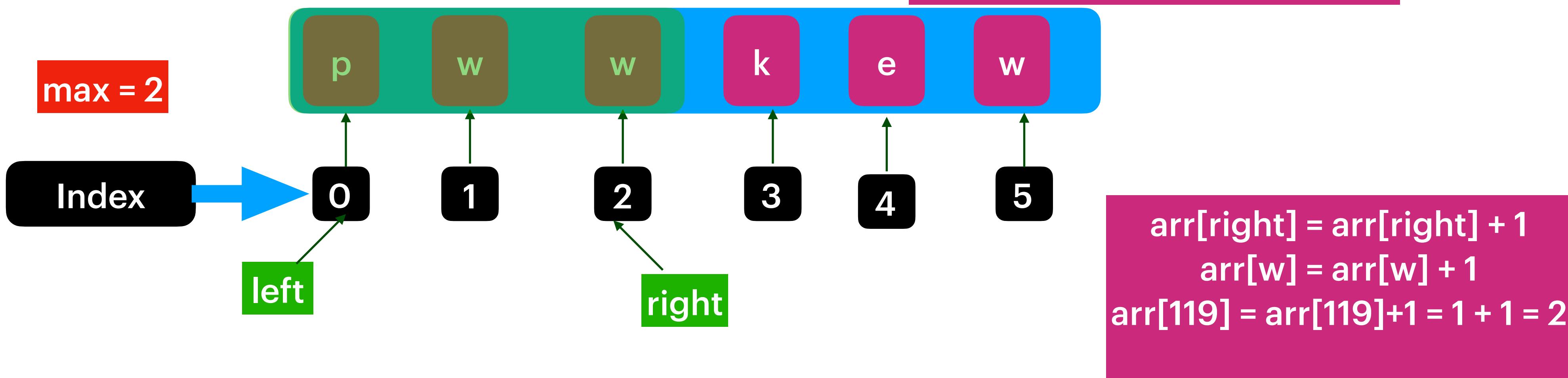
Solution is : SlidingWindow with  
repeated move of leftPointer when there is duplicate character

..... Take the int[] array of size 128

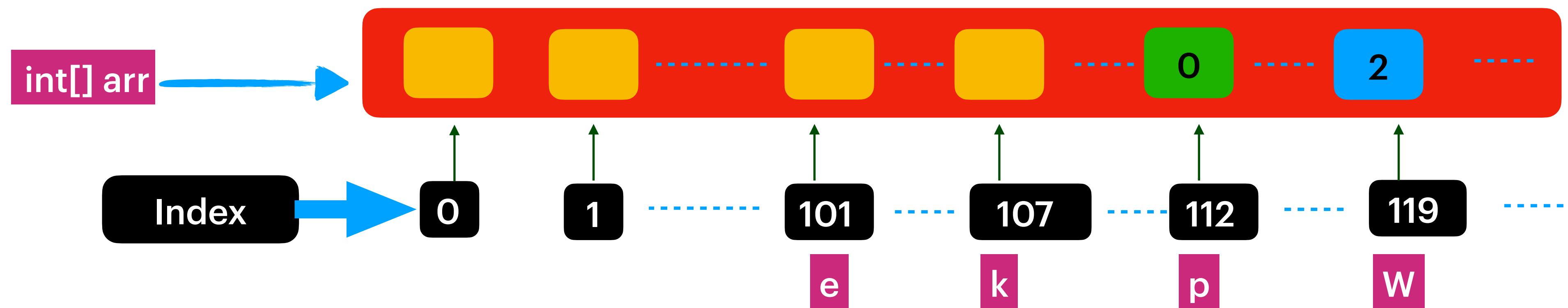
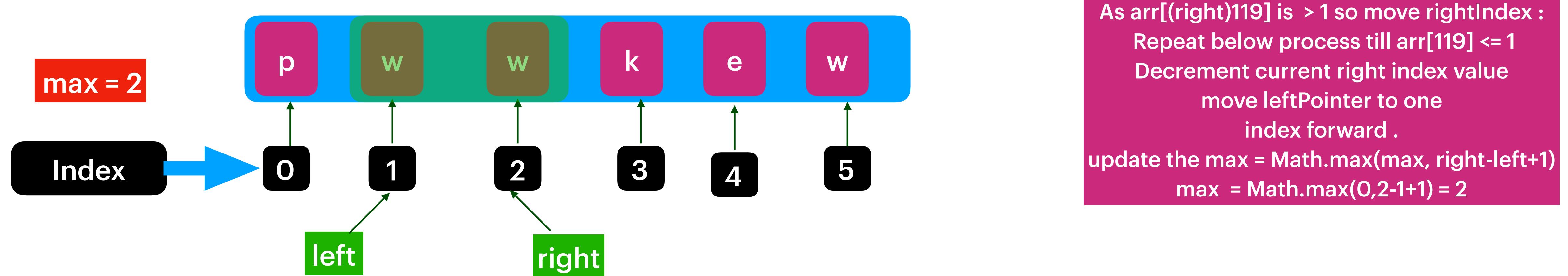
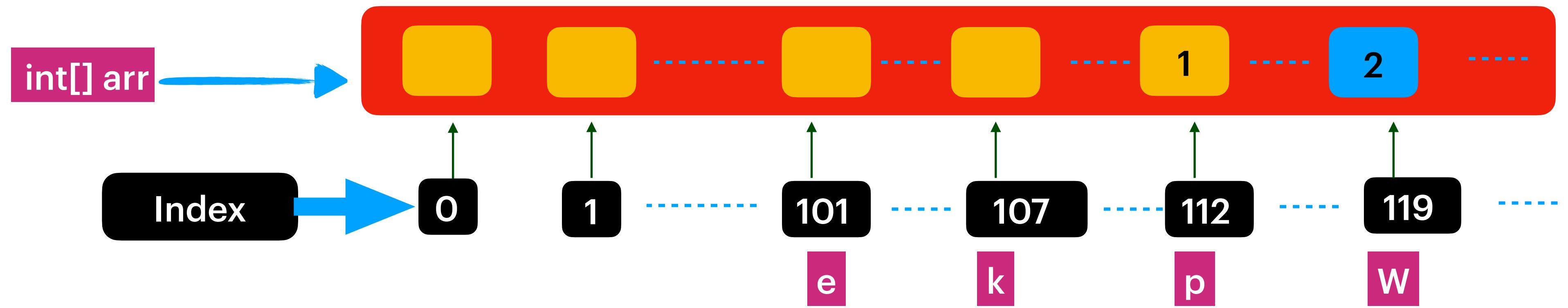


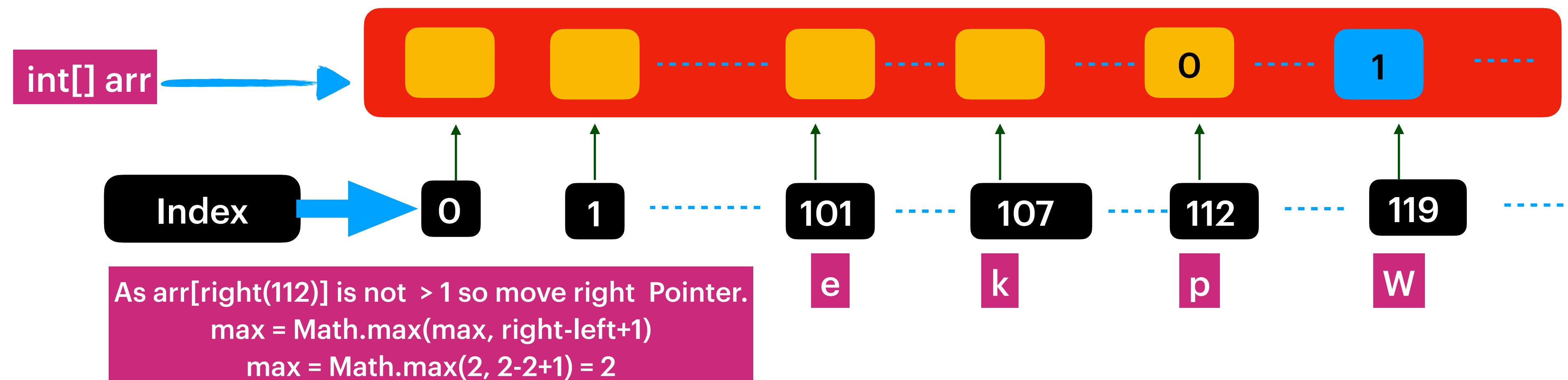
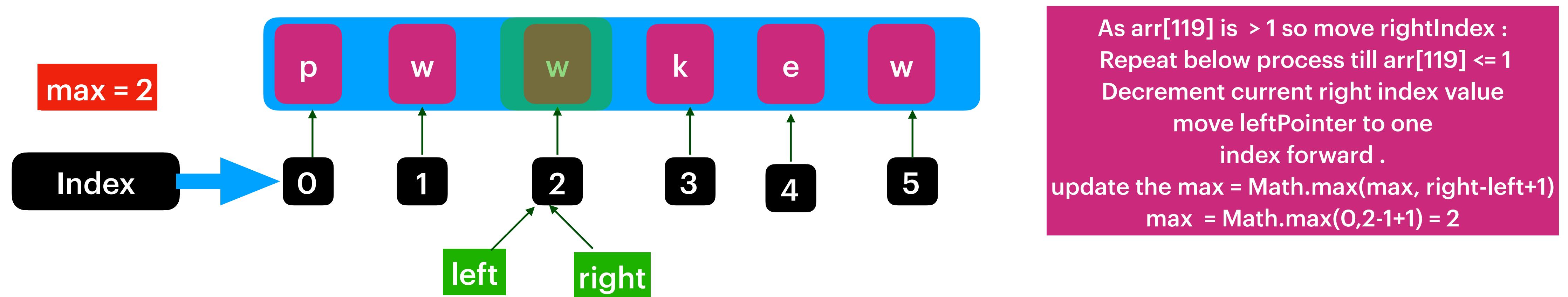
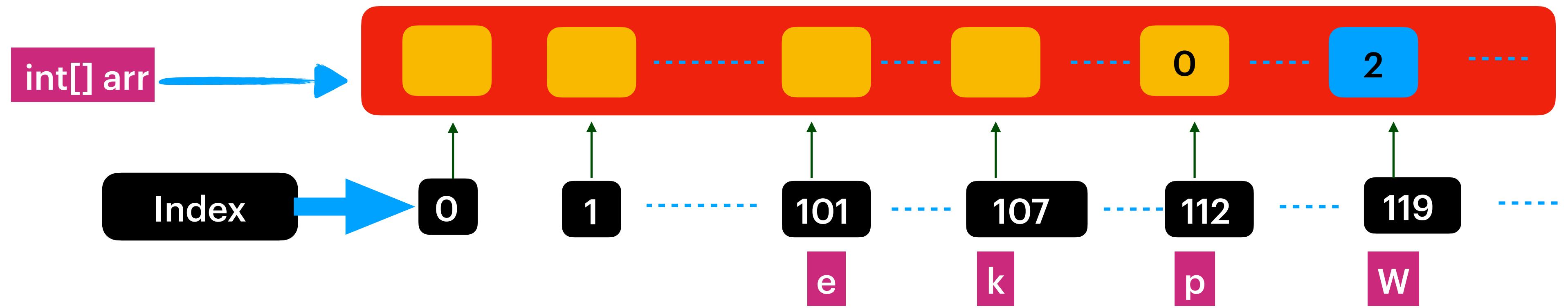
Solution is : SlidingWindow with  
repeated move of leftPointer when there is duplicate character

..... Take the int[] array of size 128



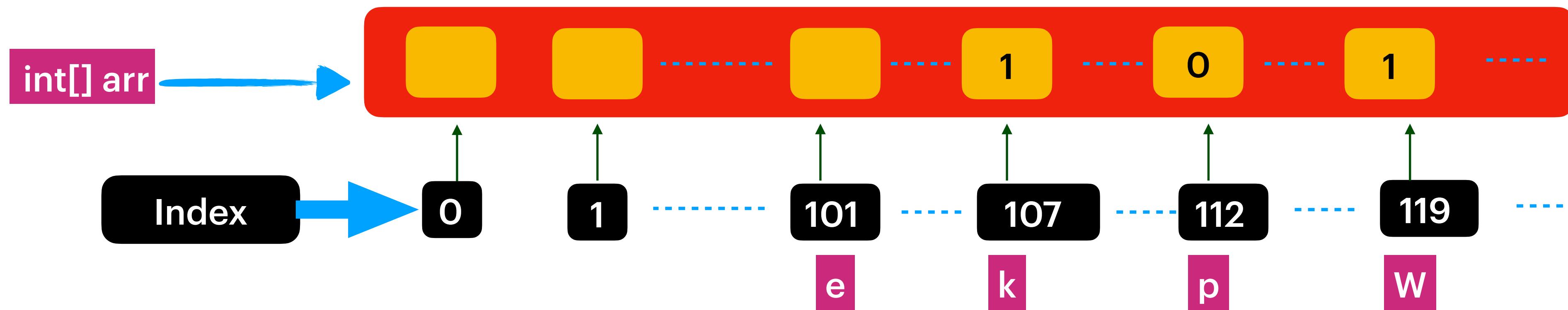
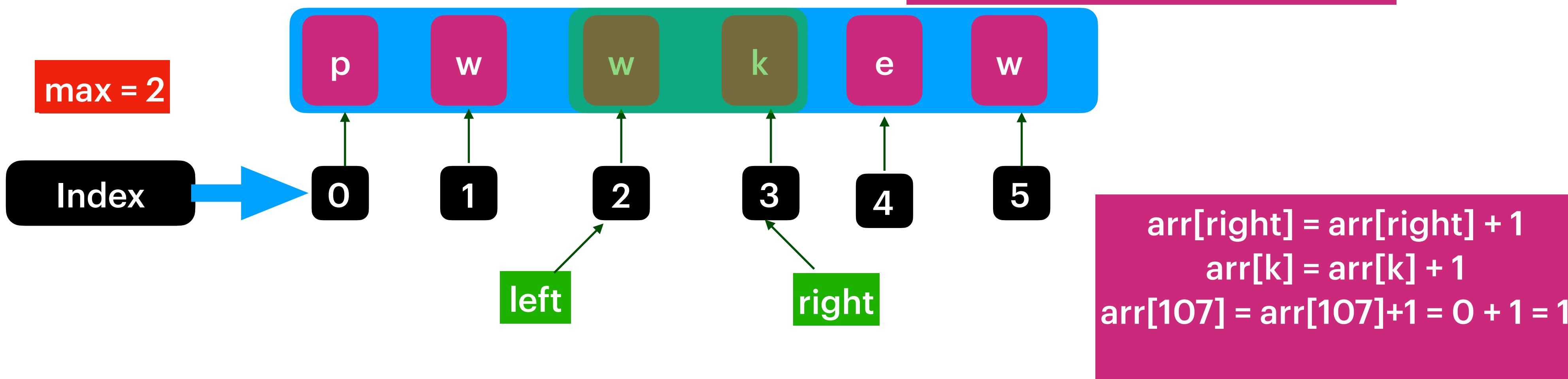
As  $\text{arr}[119](\text{right})$  is  $> 1$  Repeat below process till  $\text{arr}[119] \leq 1$   
Decrement current left Pointer value  
forward the leftPointer to one index forward





Solution is : SlidingWindow with  
repeated move of leftPointer when there is duplicate character

..... Take the int[] array of size 128



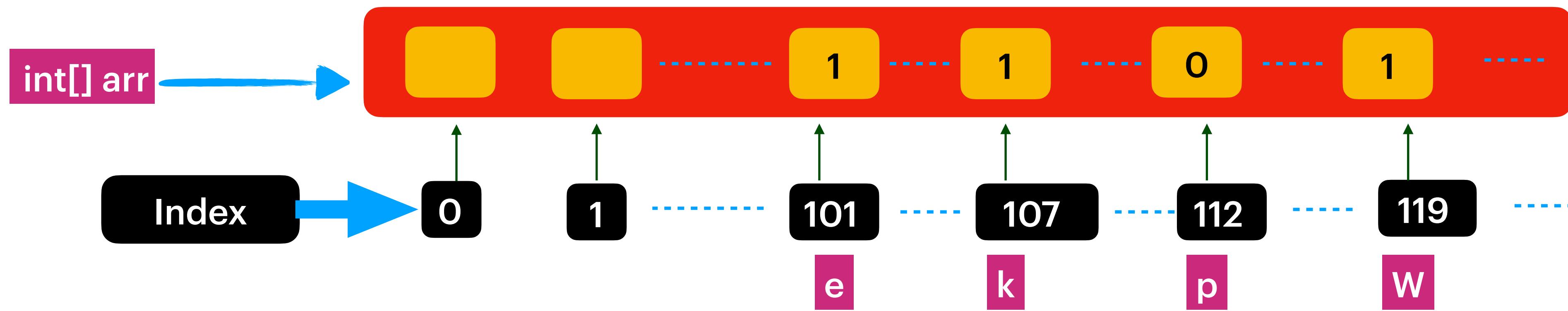
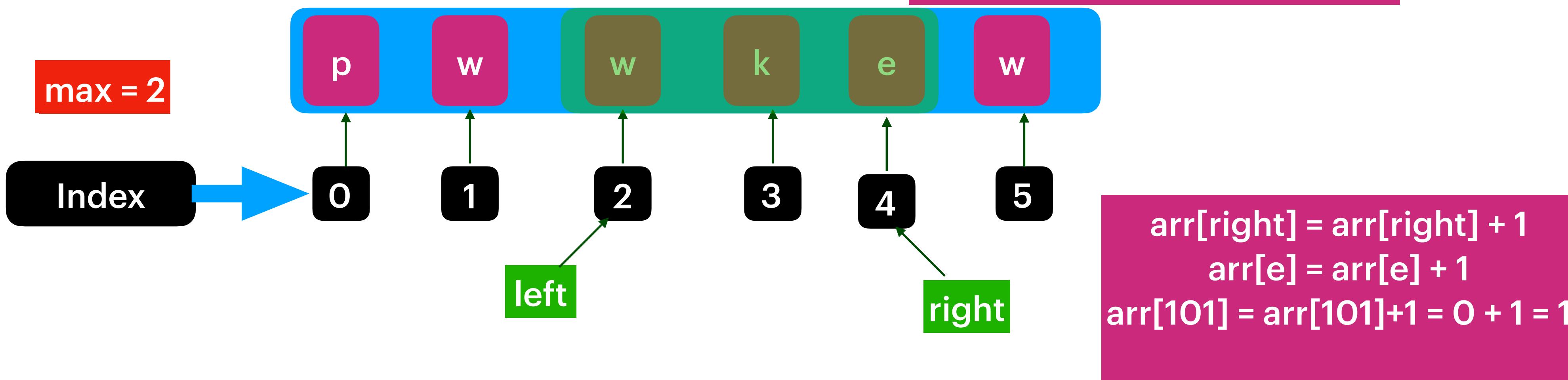
As arr[107(k)] is not > 1 so move rightIndex

$\text{max} = \text{Math.max}(\text{max}, \text{right}-\text{left}+1)$

$\text{max} = \text{Math.max}(2, 3-2+1) = 2$

Solution is : SlidingWindow with  
repeated move of leftPointer when there is duplicate character

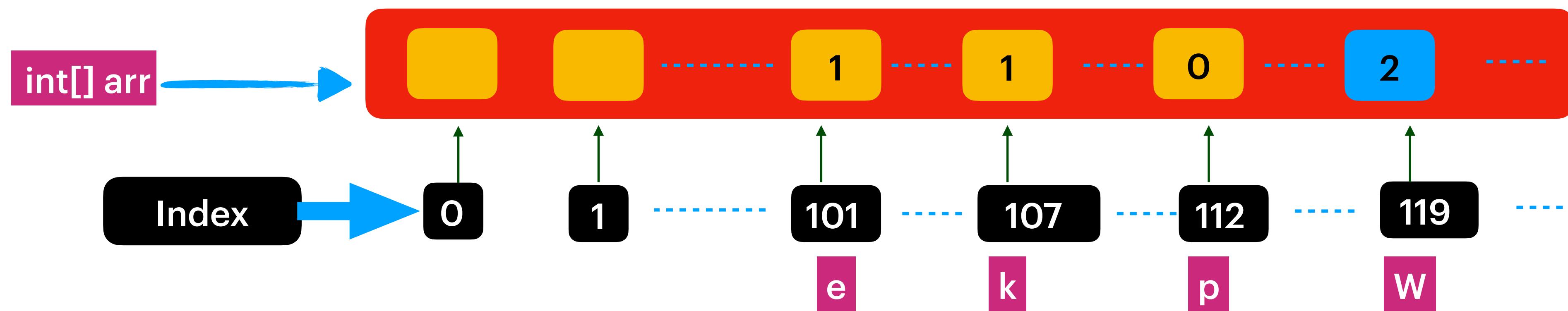
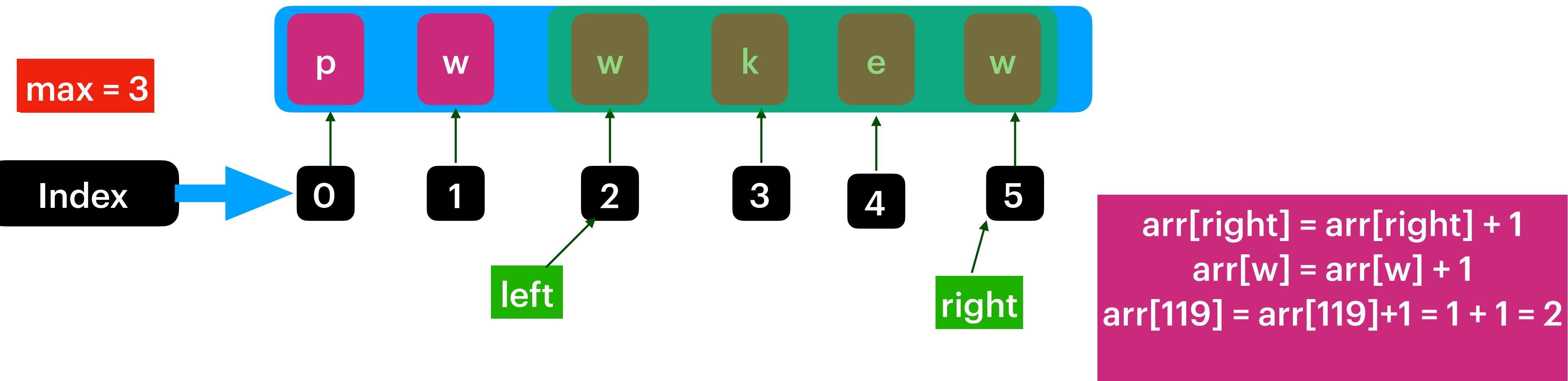
..... Take the int[] array of size 128



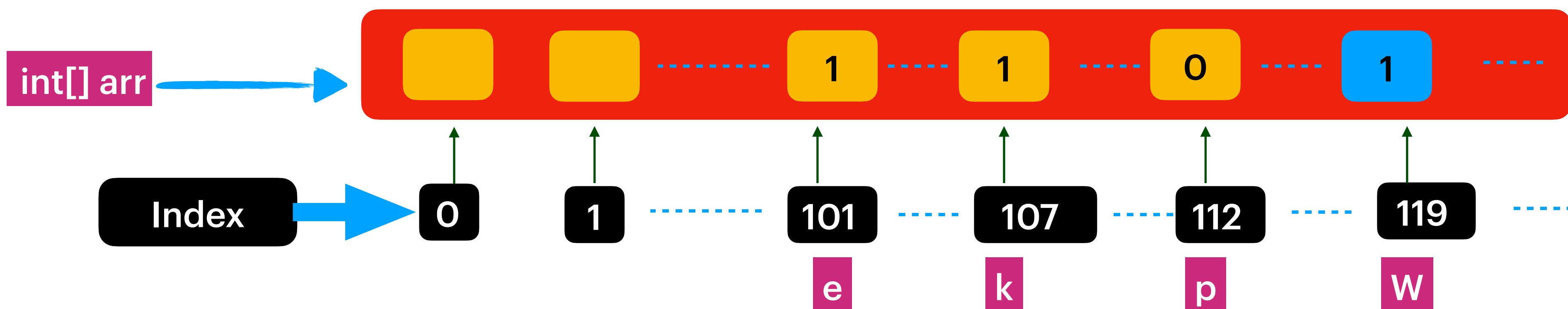
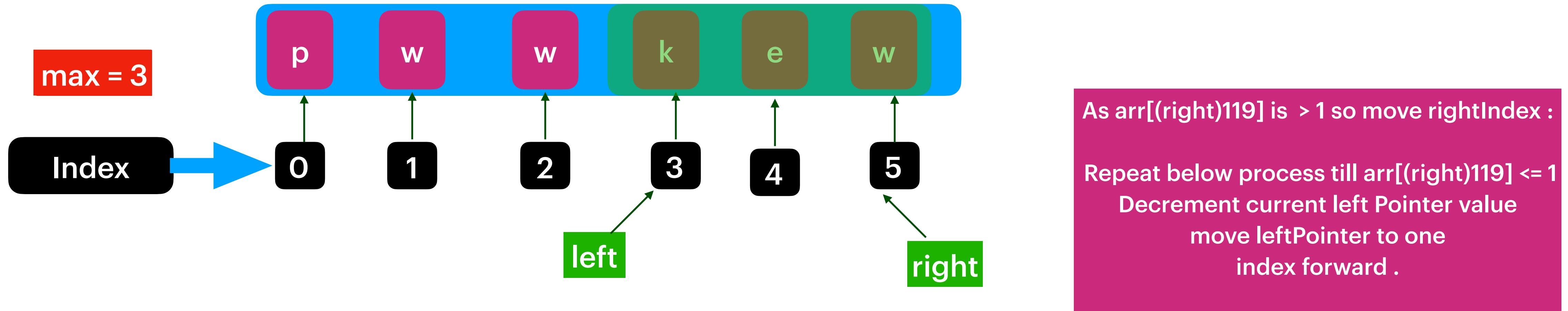
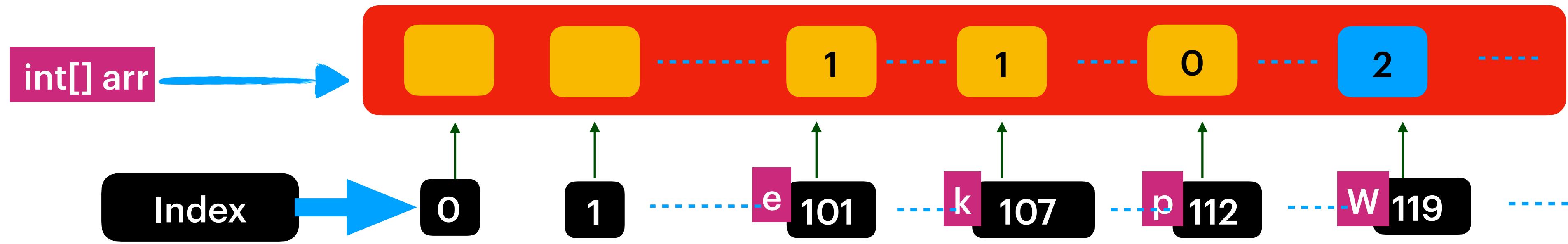
As  $\text{arr}[101(\text{e})]$  is not  $> 1$  so move rightIndex

$\text{max} = \text{Math.max}(\text{max}, \text{right}-\text{left}+1)$

$\text{max} = \text{Math.max}(2, 4-2+1) = 3$



As arr[119] is > 1 so move rightIndex :  
 Repeat below process till arr[119] <= 1  
 Decrement current right index value  
 move leftPointer to one  
 index forward .

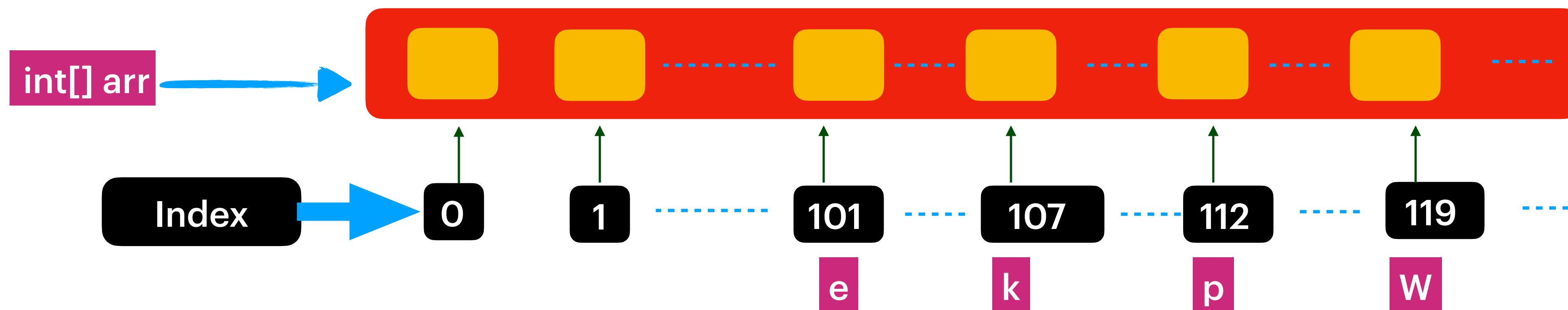
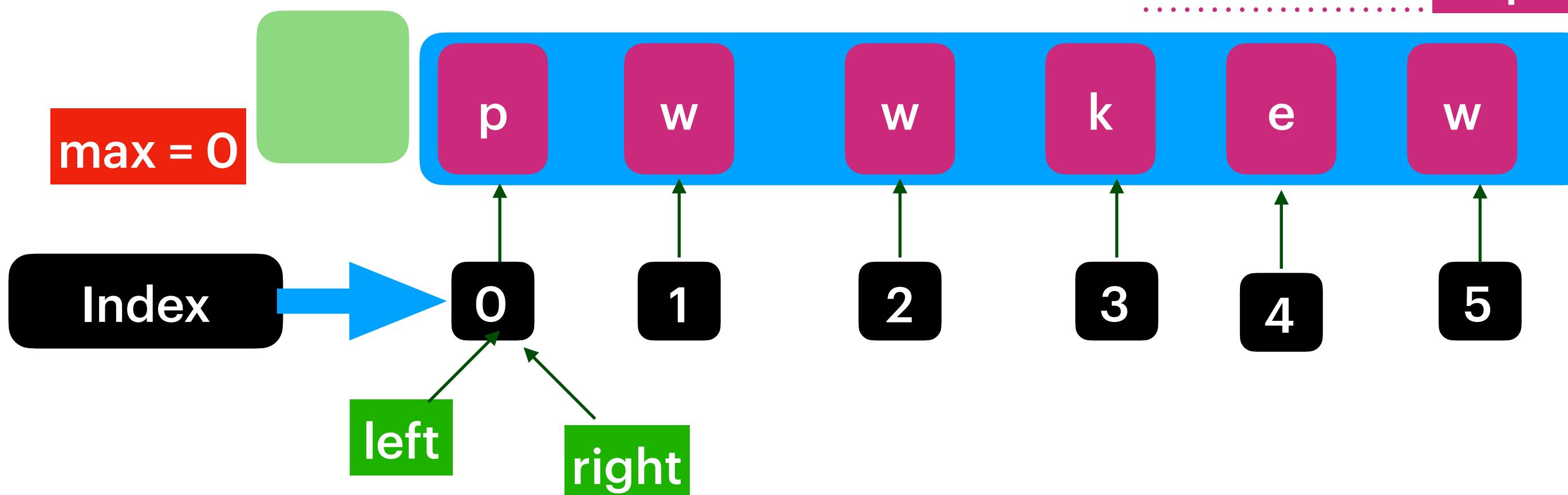


As arr[119] is not > 1 so move rightIndex  
update the  $\text{max} = \text{Math.max}(\text{max}, \text{right-left}+1)$   
 $\text{max} = \text{Math.max}(3, 5-3+1) = 3$

**ServiceNow !!!**

Solution is : SlidingWindow with  
repeated move of leftPointer when there is duplicate character

Take the int[] array of size 128  
Represents all the characters ASCII values



## How Many Numbers Are Smaller Than the Current Number?

Given the array `nums`, for each `nums[i]` find out how many numbers in the array are smaller than it. That is, for each `nums[i]` you have to count the number of valid `j`'s such that  $j \neq i$  and  $\text{nums}[j] < \text{nums}[i]$ .

Return the answer in an array.

Input: `nums = [8,1,2,2,3]`

Output: `[4,0,1,1,3]`

Explanation:

For `nums[0]=8` there exist four smaller numbers than it (1, 2, 2 and 3).

For `nums[1]=1` does not exist any smaller number than it.

For `nums[2]=2` there exist one smaller number than it (1).

For `nums[3]=2` there exist one smaller number than it (1).

For `nums[4]=3` there exist three smaller numbers than it (1, 2 and 2).

Input: `nums = [6,5,4,8]`

Output: `[2,1,0,3]`

Constraints :  
 $2 \leq \text{nums.length} \leq 500$   
 $0 \leq \text{nums}[i] \leq 100$

Input: `nums = [7,7,7,7]`

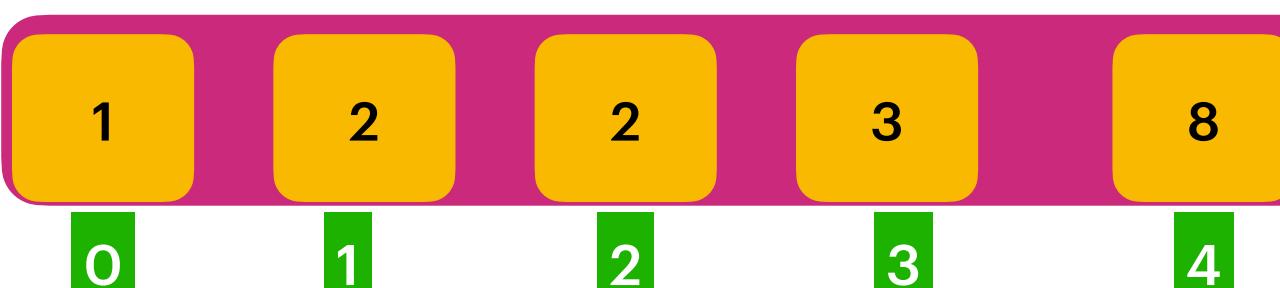
Output: `[0,0,0,0]`

`[8,1,2,2,3]`

Sort

`{1,2,2,3,8}`

Smallest number of elements less the each element is  
equals to  
minimum index number of a given element.



**inputArr:: [8,1,2,2,3,8,8] expected Output:: [4,0,1,1,3,4,4]**

=>Time Complexity:

=> O(n) copy to Output Arr + nlog(n) sort + O(n) lookUpMap + O(n) copyResult to Output Array

=> O(3n) + nlog(n) = nlog(n)

Space Complexity:

=>O(n) copy + O(n) map = O(2n) = O(n)

**outPutArr(copy) :: [8,1,2,2,3,8,8]**

**nlog(n) nums :: [1,2,2,3,8,8,8]**

**map ::**

1 -> 0

2 -> 1

3 -> 3

8 -> 4

Each element is mapped to its smallest index number.

**outPutArr(copy) :: [8,1,2,2,3,8,8]**  
**traverse [4,0,1,1,3,4,4]**

## Design Unique Word Abbreviation

The abbreviation of a word is a concatenation of its first letter, the number of characters between the first and last letter, and its last letter. If a word has only two characters, then it is an abbreviation of itself.

For example:

dog --> d1g because there is one letter between the first letter 'd' and the last letter 'g'.

internationalization --> i18n because there are 18 letters between the first letter 'i' and the last letter 'n'.

it --> it because any word with only two characters is an abbreviation of itself.

Implement the ValidWordAbbr class:

**ValidWordAbbr(String[] dictionary)** Initializes the object with a dictionary of words.

**boolean isUnique(string word)** Returns true if either of the following conditions are met (otherwise returns false):

There is no word in dictionary whose abbreviation is equal to word's abbreviation.

For any word in dictionary whose abbreviation is equal to word's abbreviation, that word and word are the same.

### Input

```
["ValidWordAbbr", "isUnique", "isUnique", "isUnique", "isUnique", "isUnique", "isUnique"]
[[["deer", "door", "cake", "card"]], ["dear"], ["cart"], ["cane"], ["make"], ["cake"]]
```

### Output

```
[null, false, true, false, true, true]
```

### Explanation

```
ValidWordAbbr validWordAbbr = new ValidWordAbbr(["deer", "door", "cake", "card"]);
```

```
validWordAbbr.isUnique("dear"); // return false, dictionary word "deer" and word "dear" have the same abbreviation "d2r" but are not the same.
```

```
validWordAbbr.isUnique("cart"); // return true, no words in the dictionary have the abbreviation "c2t".
```

```
validWordAbbr.isUnique("cane"); // return false, dictionary word "cake" and word "cane" have the same abbreviation "c2e" but are not the same.
```

```
validWordAbbr.isUnique("make"); // return true, no words in the dictionary have the abbreviation "m2e".
```

```
validWordAbbr.isUnique("cake"); // return true, because "cake" is already in the dictionary and no other word in the dictionary has "c2e" abbreviation.
```

### Constraints :

1 <= dictionary.length <= 3 \* 104

1 <= dictionary[i].length <= 20

dictionary[i] consists of lowercase English letters.

1 <= word.length <= 20

word consists of lowercase English letters.

Design a data structure that accepts a stream of integers and checks if it has a pair of integers that sum up to a particular value.

Implement the TwoSum class:

**TwoSum()** Initializes the TwoSum object, with an empty array initially.

**void add(int number)** Adds number to the data structure.

**boolean find(int value)** Returns true if there exists any pair of numbers whose sum is equal to value, otherwise, it returns false.

**Input**

```
["TwoSum", "add", "add", "add", "find", "find"]
[], [1], [3], [5], [4], [7]
```

**Output**

```
[null, null, null, null, true, false]
```

**Constraints:**

-105 <= number <= 105

-2<sup>31</sup> <= value <= 2<sup>31</sup> - 1

**Explanation**

```
TwoSum twoSum = new TwoSum();
```

```
twoSum.add(1); // [] --> [1]
```

```
twoSum.add(3); // [1] --> [1,3]
```

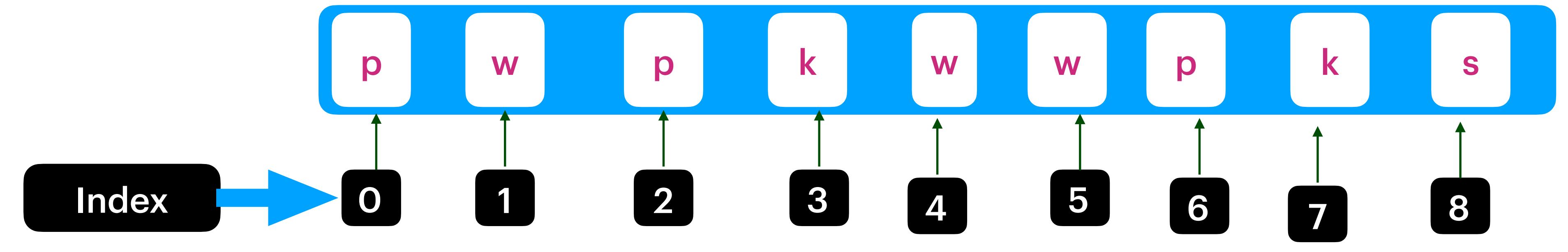
```
twoSum.add(5); // [1,3] --> [1,3,5]
```

```
twoSum.find(4); // 1 + 3 = 4, return true
```

```
twoSum.find(7); // No two integers sum up to 7, return false
```



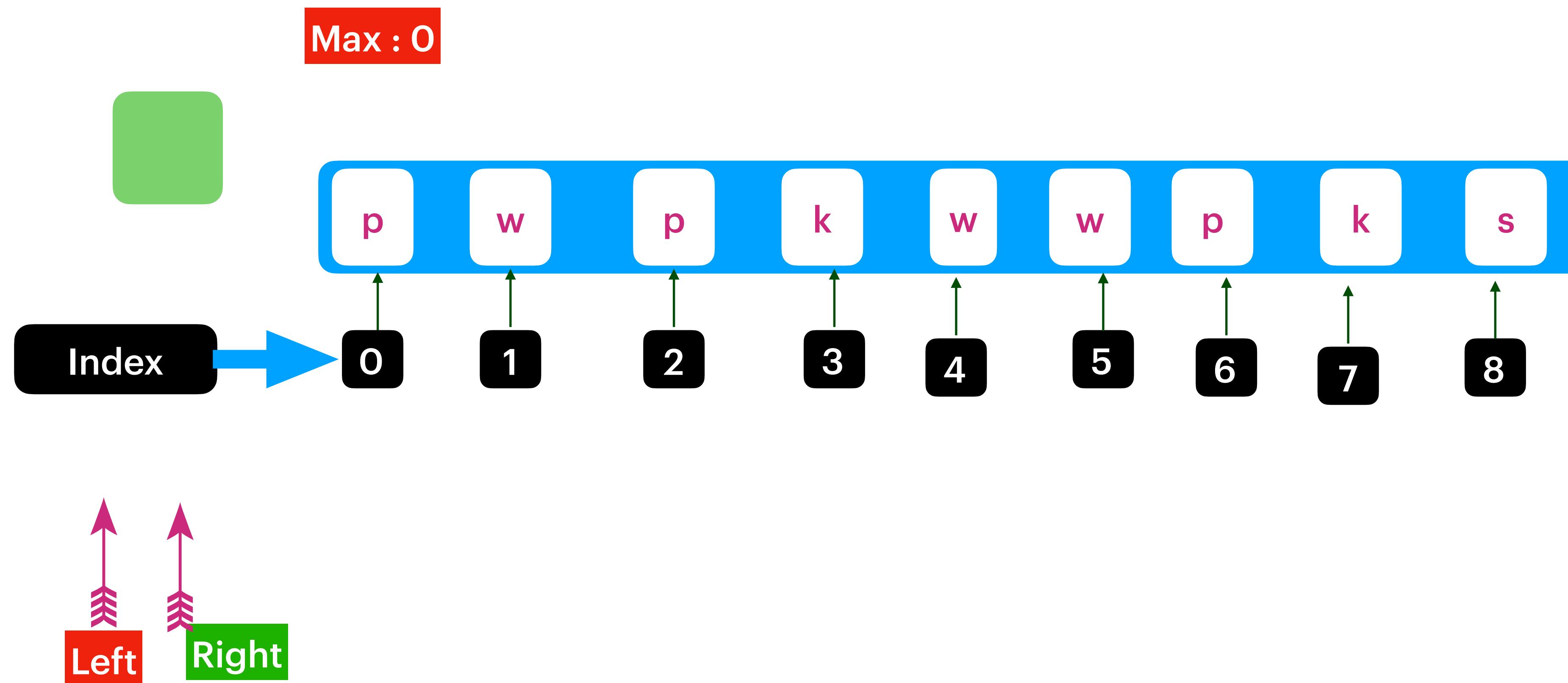
## Longest Substring Without Repeating Characters.



Output : 4. => wpks

## Longest Substring Without Repeating Characters.

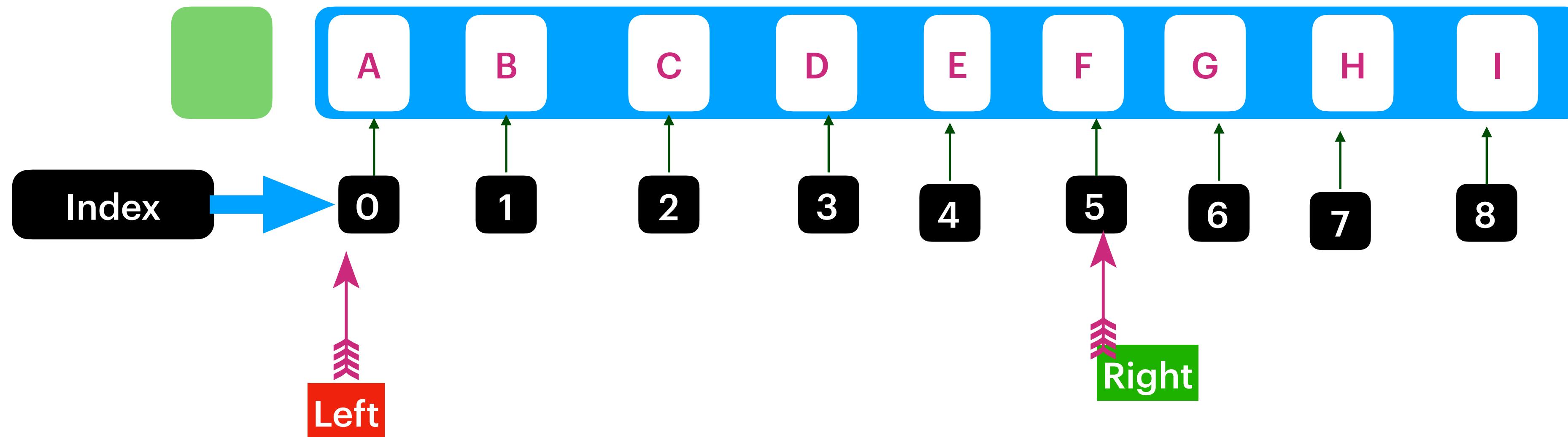
Output : 4. => wpks



# Longest Substring Without Repeating Characters.

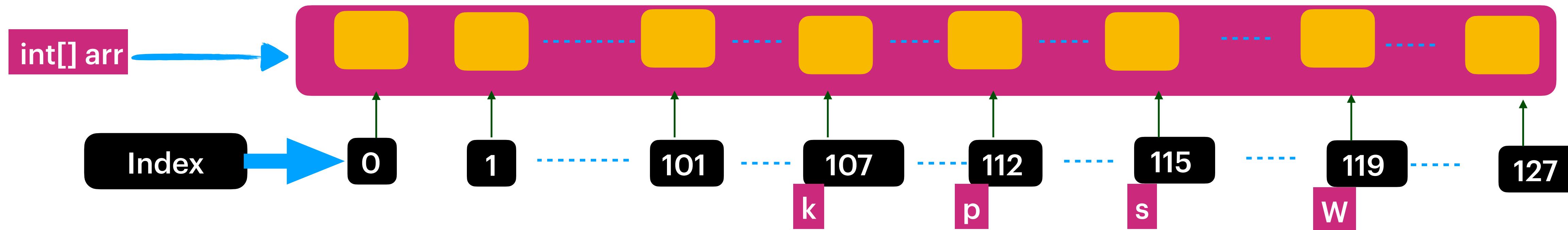
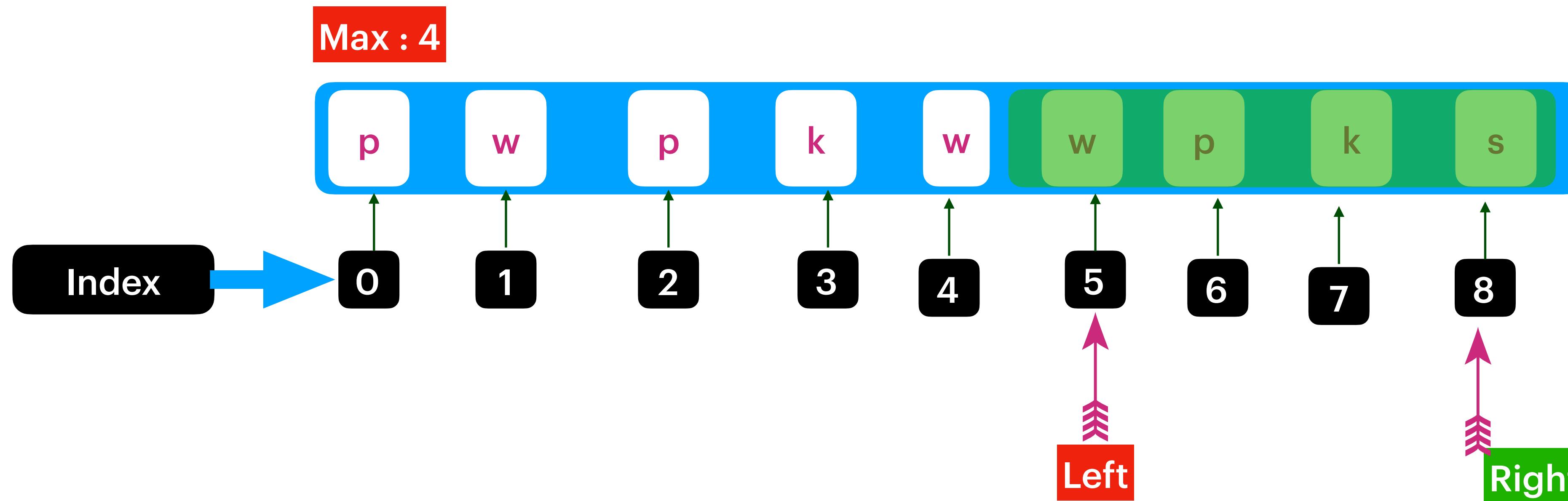
Max :0

Output : 9. => ABCDEFGHI



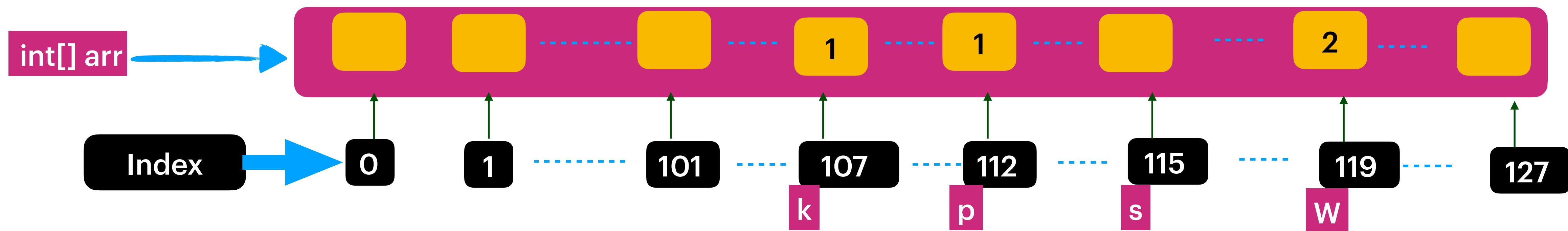
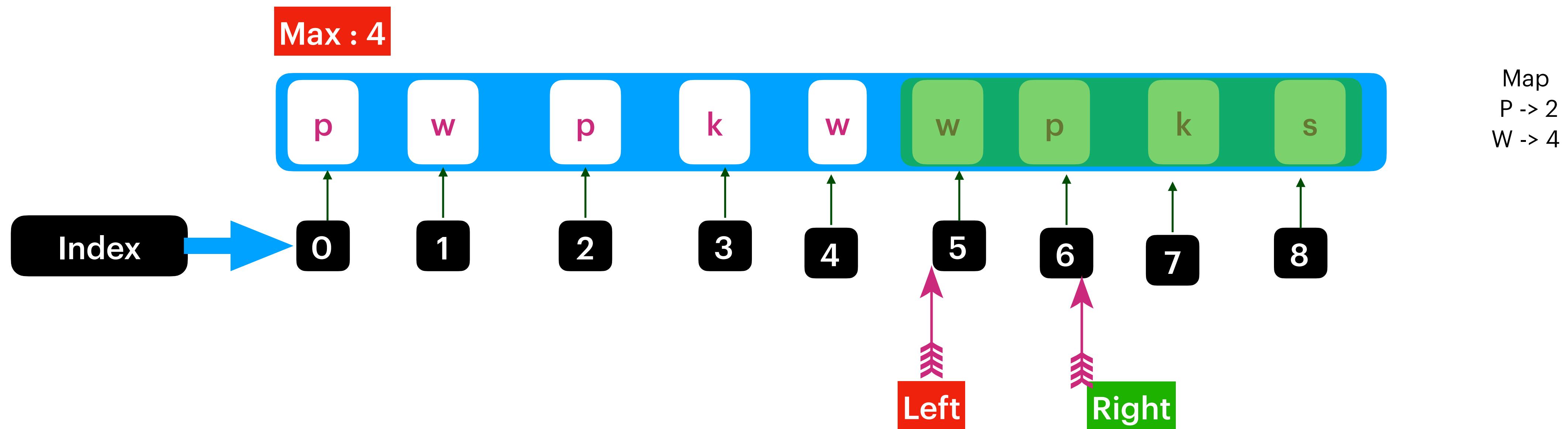
## Longest Substring Without Repeating Characters.

Output : 4. => wpks



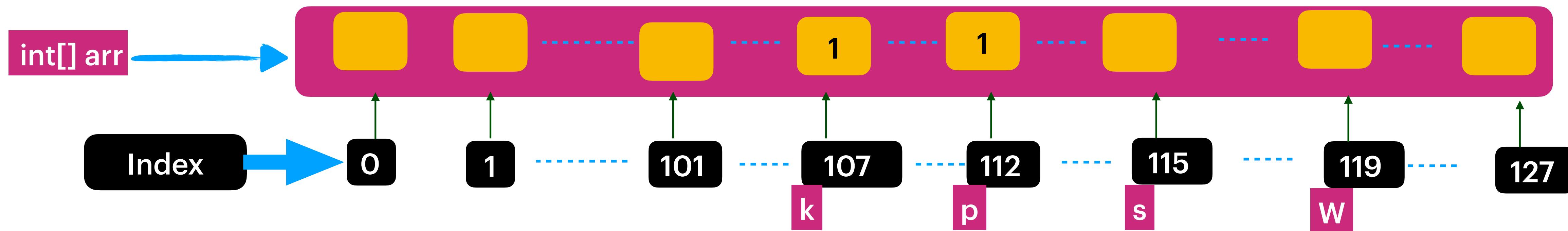
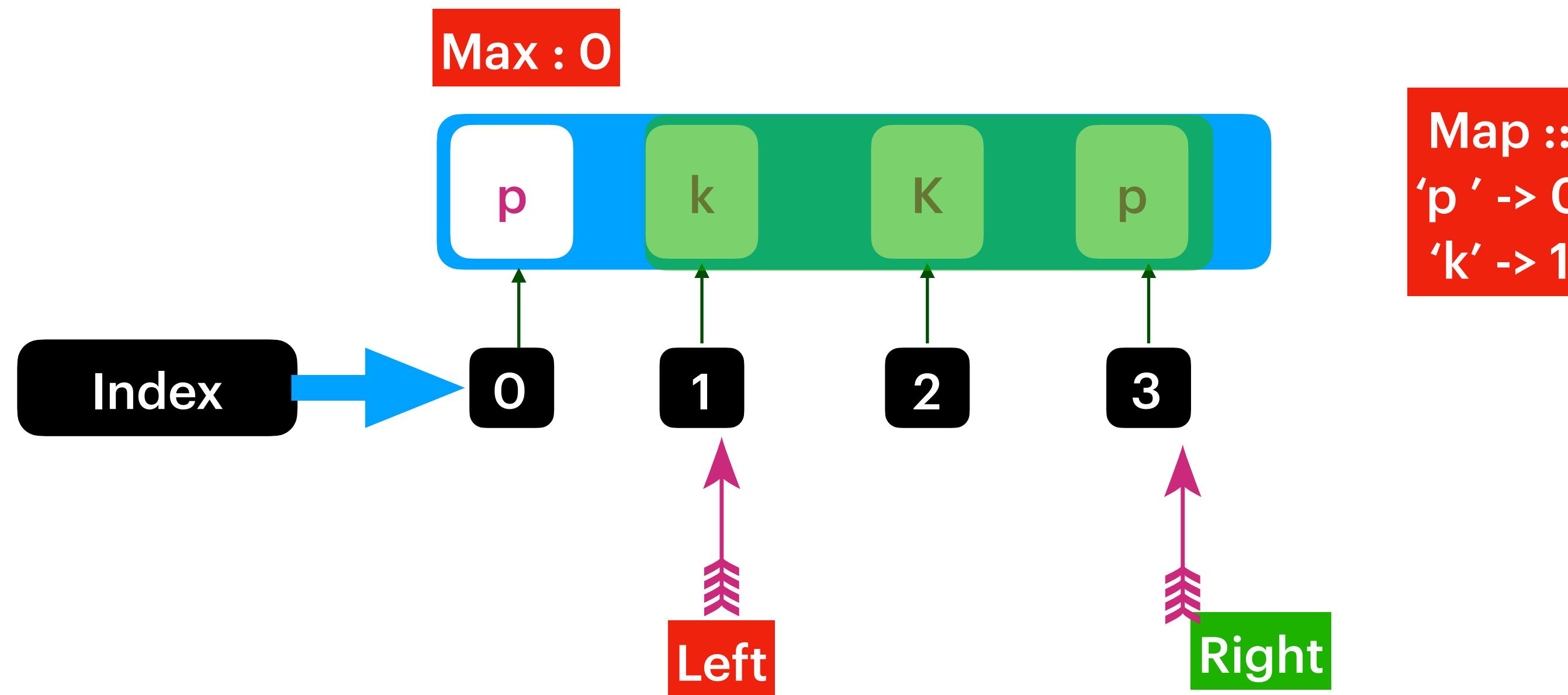
## Longest Substring Without Repeating Characters.

Output : 4. => wpks



## Longest Substring Without Repeating Characters.

Output : 4. => wpks



## Design Two Sum in infinite Stream

Example :  
add(3);  
add(1);  
add(2);

Stream

Adding data Streams , allows duplicates  
Contains +ve & -ve numbers

public void add(int value)

Constraints  
Average Case of Time Complexity :  
add(int) => O(1)  
find(int) => O(n)

adds data to the Stream

public boolean find(int val)

returns true if two sum exists in stream  
with the given value.

find(3) => true (onStream 1+2)  
find(2) => false (no sum of two elements on Stream)

Example :  
add(0);  
add(1);  
add(0);

3,2,1,0,1,0

find(0) => true (onStream 0+0)  
find(2) => true (on Stream 1+1)

3,2,1,0,1,0,-1,6

Example :  
add(-1);  
add(6);

find(5) => true  
onStream 6-1 = 5  
find(8) => true 6+2  
find(10) => false  
(6+3+1 = 10 but its not  
Two element sum)

## Brute Force Solution

Stream Storage : ArrayList

Input : (3,2,1)

find(val)

find(3):

Find Diff in each index (n)

Check diff present in array or not (n)

Time Complexity :  $O(n^2)$

## Sorting Solution

Stream Storage : ArrayList

Input : (3,2,1)

Sorting : find(val)

Time Complexity :  $n \log(n) + O(n/2) = n \log(n)$

Sort (1,2,3)

Lets do Search Pattern:  
low = index0 = 0  
high = n-1 = 2

Find twoSum with  $\text{arr}[low] + \text{arr}[high]$   
 $\text{twoSum} > \text{val}$   
 $\text{high} = \text{high}-1$   
 $\text{twoSum} < \text{val}$   
 $\text{low} = \text{low}+1$   
 $\text{twoSum} == \text{val}$   
return true.

## Hashing Solution

Stream Storage : Map  
Input : (3,2,1,3)  
find(val)

Time Complexity :  $O(n) + O(1)/O(\log n) = O(n)$

Map:<key , value>  
3 -> 2  
2 -> 1  
1 -> 1

find(5)

Map < key (element), value (repeatedCount >

Iterate the map and find the diff at each iteration.

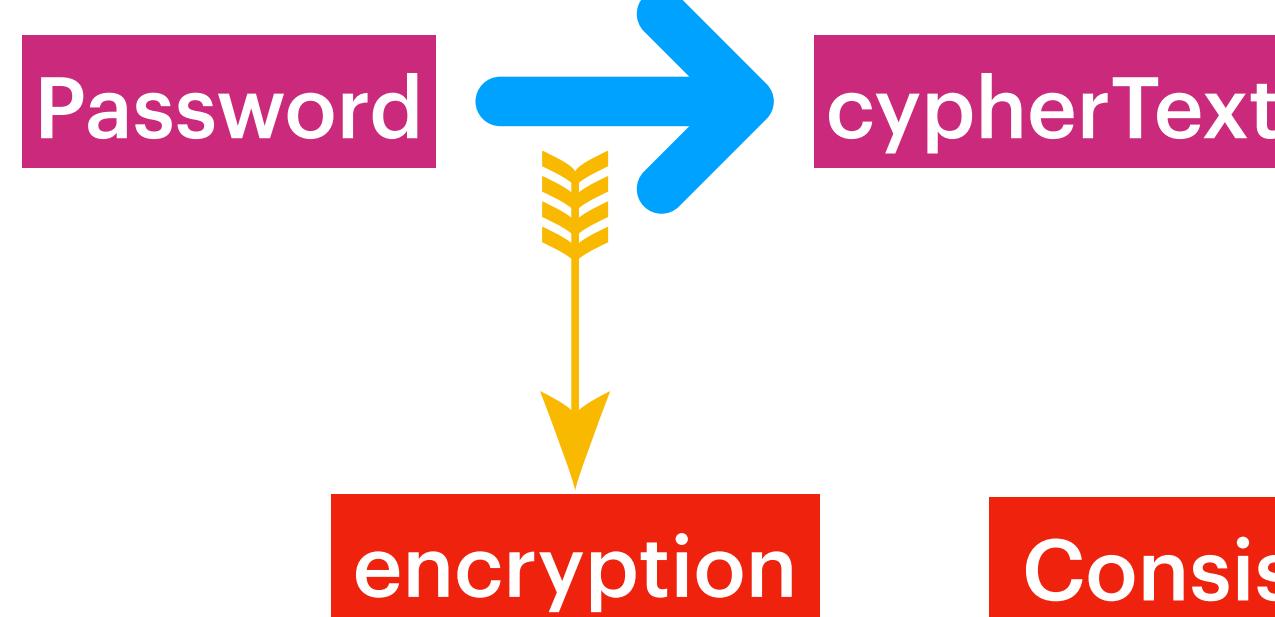
If the diff is not equals to currentKey then  
check does the diff exists in map or not.

If the diff is equals to currentKey then check  
Does the currentKey repeatedCount is > 1 or not.

Iterate the map and find the diff at each iteration.

If the diff is not equals to currentKey then  
check does the diff exists in map or not.

If the diff is equals to currentKey then check  
Does the currentKey repeatedCount is > 1 or not.



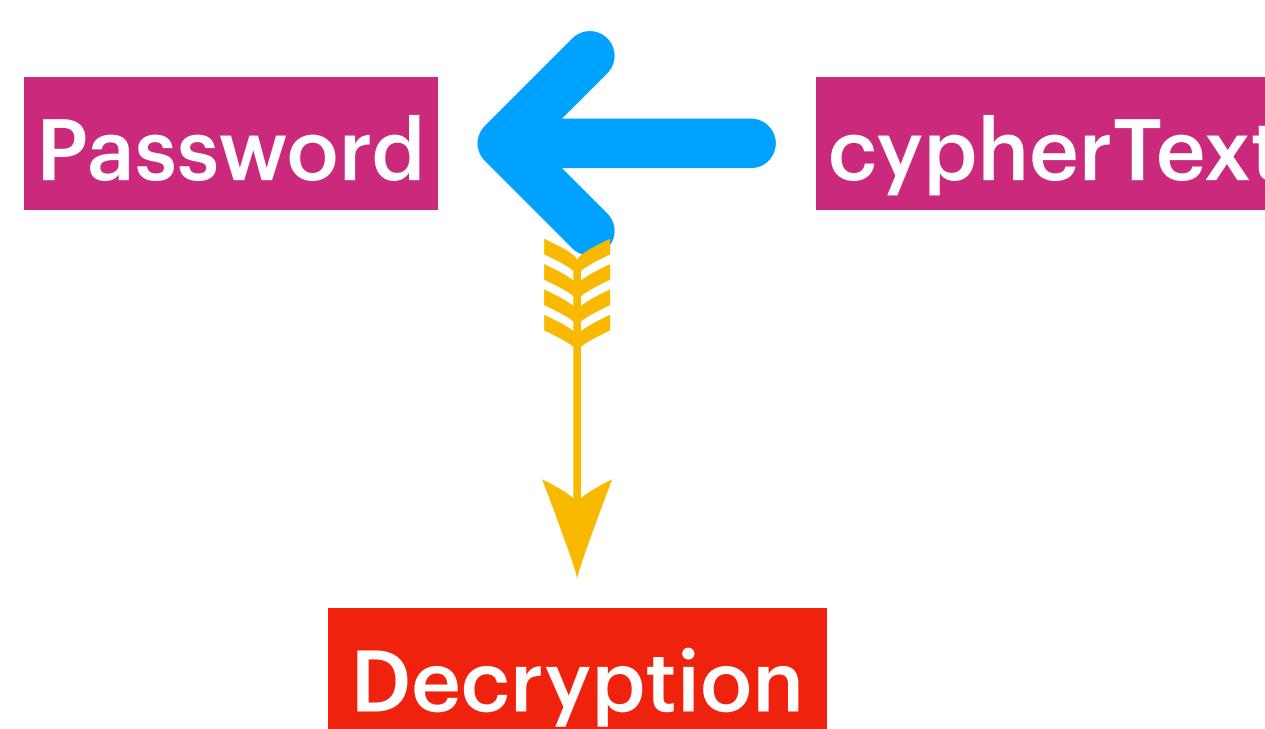
Randomisation

Uniform ✓ 2,1,4,3

Periodic ✓ 2,1,4,3 2,1,4,3 2,1,4,3

Consistence/Efficiency ✓ Returns Consistence Results

Song Ids : 1,2,3,4



linear congruent generator

$$x(n+1) = ax(n) + c \text{ / Modular}$$

SampleSize represents number  
Of elements.

Take Modular, a , c are the primeNumbers  
a & c < SampleSize

seedValue x(0), should be one of the value in the Sample

$$Y = mx + c$$

To avoid closest periodic in Randomisation,  
make sure moduler is going to be the closest primeNumber of  
SampleSize & a, c are also be the prime numbers.

$$x(n+1) = a x(n) + c / \text{Modular}$$

## Randomisation

$a = 2, c = 3, x(0) = 1, \text{Modular} = 5$

Uniform ✓ 2,1,4,3

Song Ids : 1,2,3,4,5

n=0

$$\begin{aligned} x(0+1) &= 2 * x(0) + 3 \% 4 = 2 * 1 + 3 \% 5 = \\ &5 \% 5 = 0 \\ x(1) &= 0 \end{aligned}$$

Periodic ✓ 2,1,4,3 2,1,4,3 2,1,4,3

Consistence ✓ Returns Consistence Results

n=1

$$\begin{aligned} x(1+1) &= 2 * x(1) + 3 \% 5 \\ &= 2 * 0 + 3 \% 5 = 3 \\ x(2) &= 3 \end{aligned}$$

n=2

$$\begin{aligned} x(2+1) &= 2 * x(2) + 3 \% 5 \\ &= 2 * 3 + 3 \% 5 = 9 \% 5 = 4 \\ x(3) &= 4 \end{aligned}$$

n=3

$$\begin{aligned} x(3+1) &= 2 * x(3) + 3 \% 5 \\ &= 2 * 4 + 3 \% 5 = 11 \% 5 = 1 \\ x(4) &= 1 \end{aligned}$$

Uniform ✓ 2,1,4,3 2,1,4,3 2,1,4,3

Consistence ✓ Returns Consistence Results

linear congruential generator

$$\dots \dots \dots x(n+1) = a x(n) + c / \text{Modular}$$

SampleSize represents number  
Of elements.

a & c < SampleSize

seedValue x(0)= should be one of value in the Sample

Finally 0,3,4,1 0,3,4,1



.....

n=4

$$\begin{aligned} x(4+1) &= 2 * x(4) + 3 \% 5 \\ &= 2 * 1 + 3 \% 5 = 5 \% 5 = 0 \\ x(5) &= 0 \end{aligned}$$

n=5

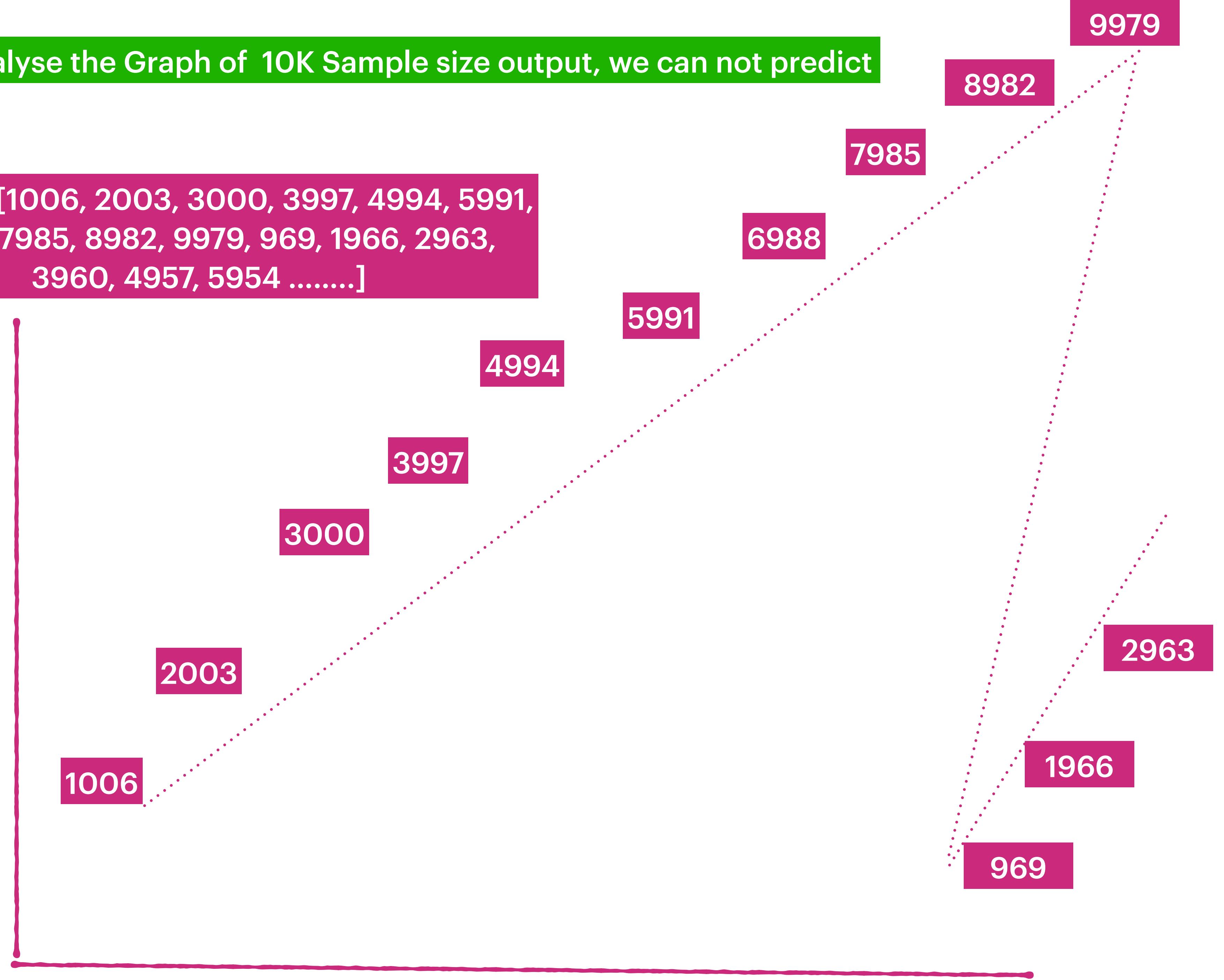
$$\begin{aligned} x(5+1) &= 2 * x(5) + 3 \% 5 \\ &= 2 * 0 + 3 \% 5 = 3 \\ x(6) &= 3 \end{aligned}$$

n=6

$$\begin{aligned} x(6+1) &= 2 * x(6) + 3 \% 5 \\ &= 2 * 3 + 3 \% 5 = 9 \% 5 = 4 \\ x(3) &= 4 \end{aligned}$$

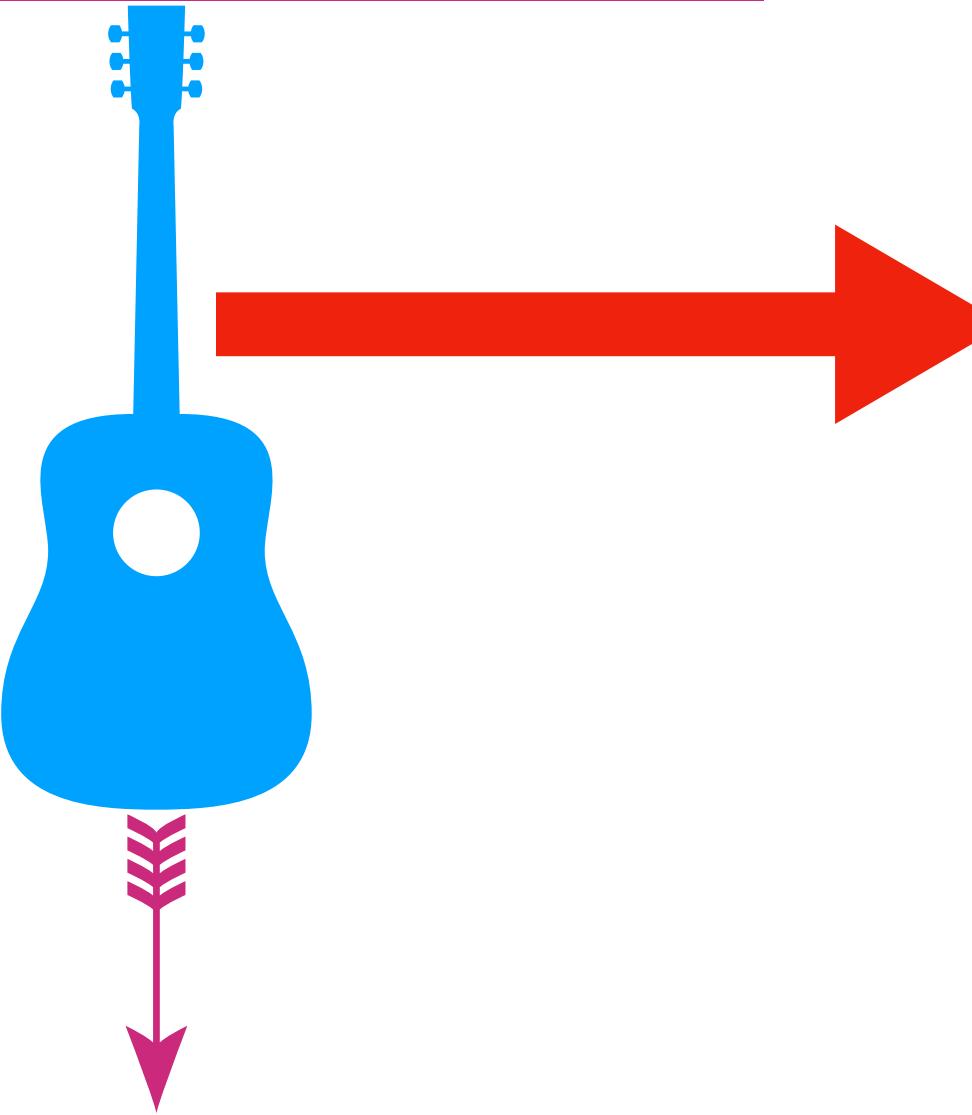
Lets Analyse the Graph of 10K Sample size output, we can not predict

Elements[1006, 2003, 3000, 3997, 4994, 5991, 6988, 7985, 8982, 9979, 969, 1966, 2963, 3960, 4957, 5954 .....]



**Constraint :: All the Operations average time complexity should be O(1)**

### Design Shuffling PlayList.



**Example :**  
addSong(7); => true  
addSong(15); => true  
addSong(3); => true  
getRandom(); => can return 7 or 15 or 3  
addSong(15) => false  
removeSong(15); => true  
getRandom(); => can return 7 or 3  
removeSong(15); => false

**public boolean addSong(int songId)**

.....  
**returns true if song added else false.**

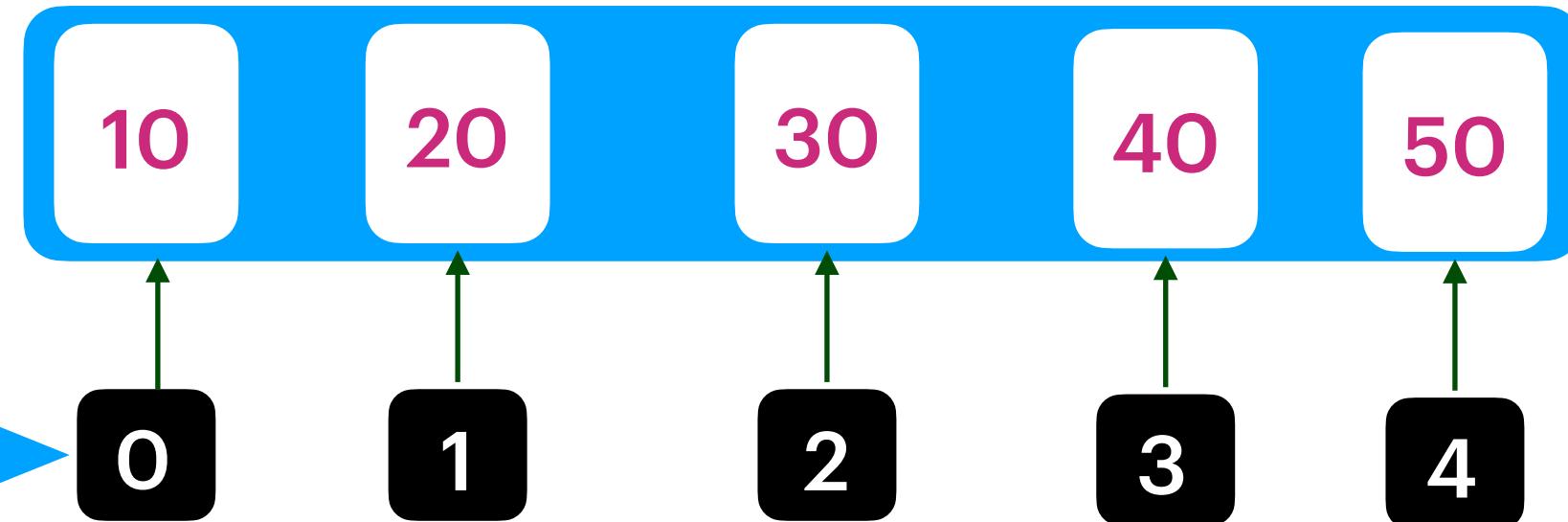
**public boolean removeSong(int songId)**

.....  
**returns true if song removed else false.**

**public boolean getRandom()**

.....  
**Should return Random song**

ArrayList : ..... 10,20,30,40,50



Add : O(1)

getRandom : O(1) ?

MapKey : SongId, Value index .....

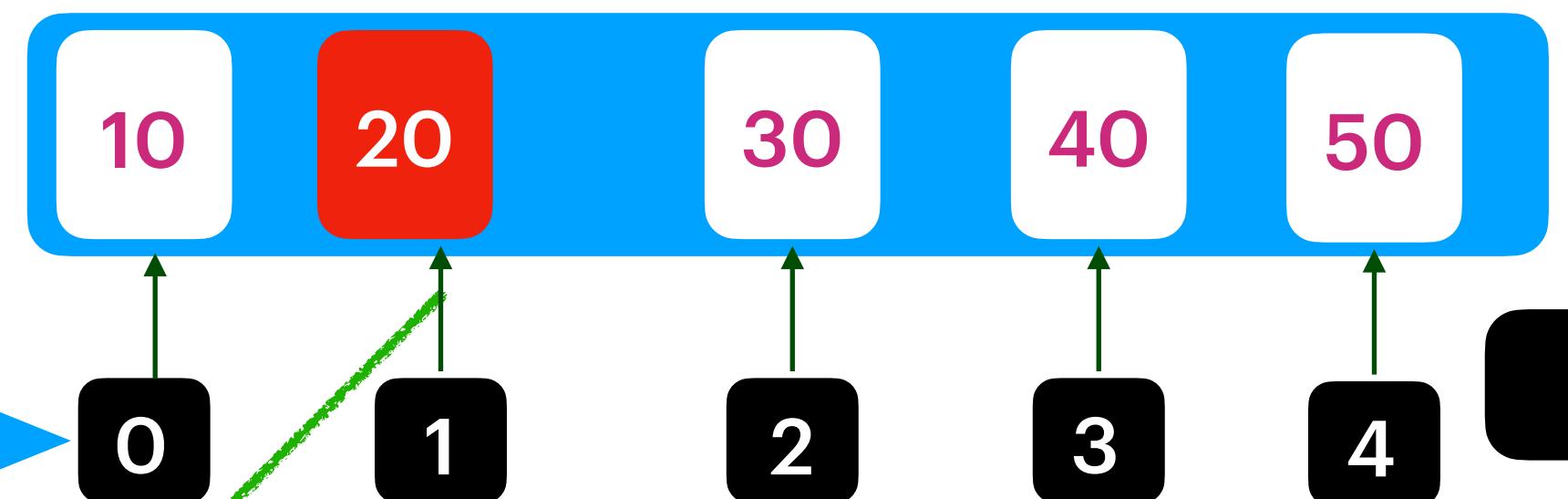
10 -> 0  
20 -> 1  
30 -> 2  
40 -> 3  
50 -> 4

Remove(20) :

MapKey : SongId, Value index

10 -> 0  
20 -> 1  
30 -> 2  
40 -> 3  
50 -> 4

Remove(20) :



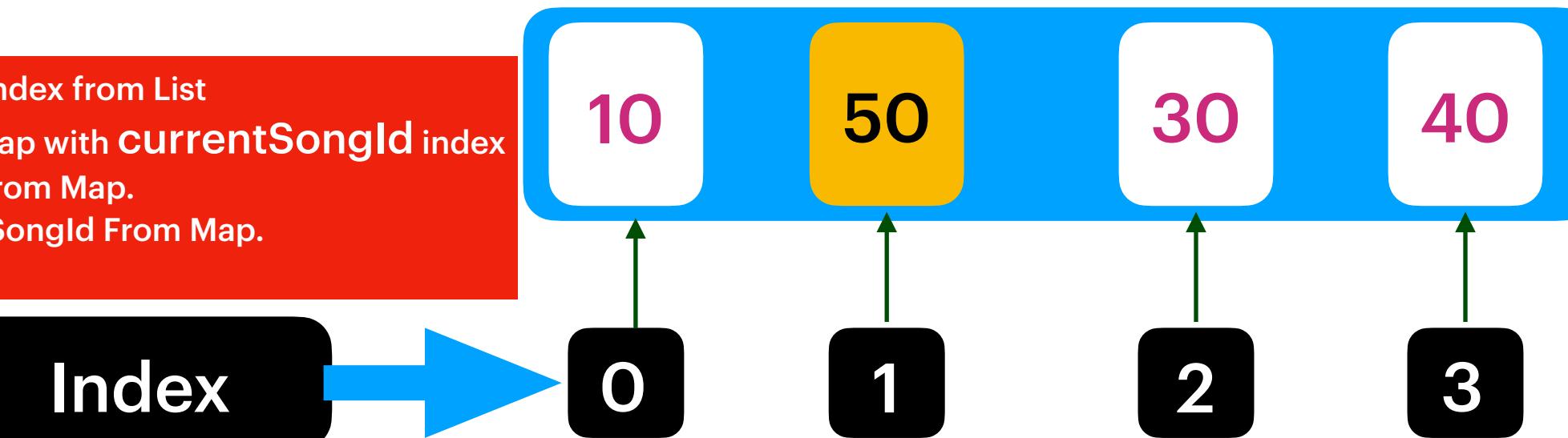
Step1 Identify songId index From Map

MapKey : SongId, Value index

10 -> 0  
30 -> 2  
40 -> 3  
50 -> 1

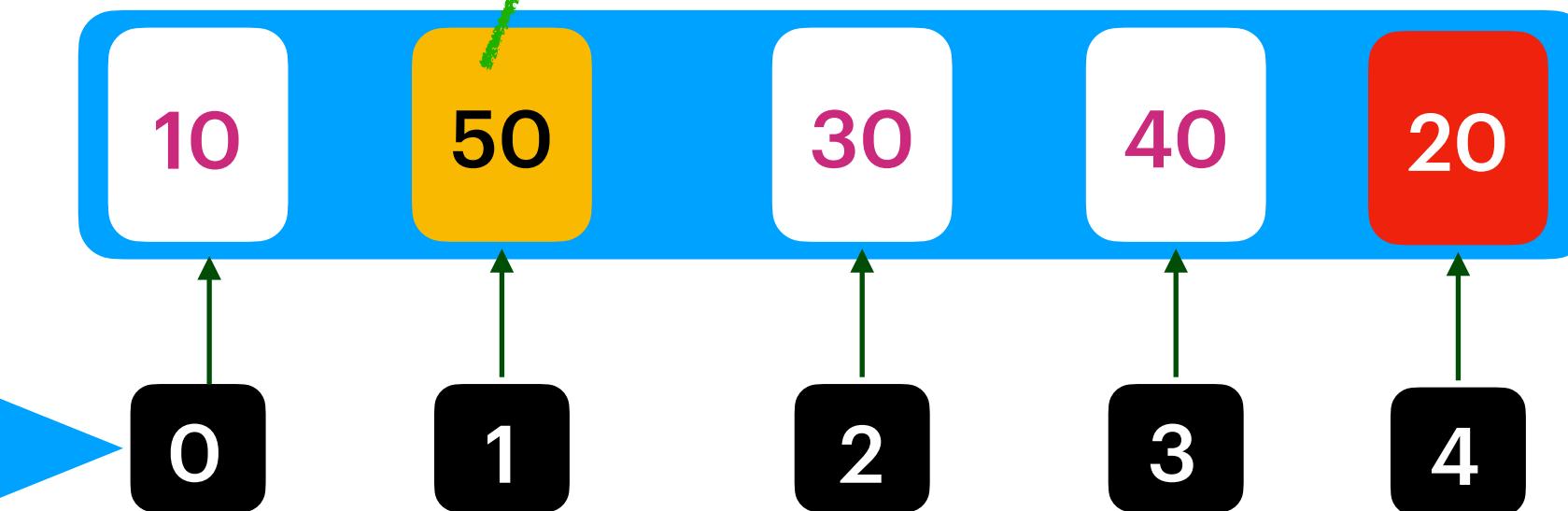
Remove lastIndex from List  
Update lastSongId index in map with currentSongId index  
songId from Map.  
Remove CurrentSongId From Map.

Step3

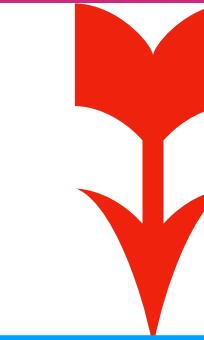


Swap with lastIndex value

Step2



## Design LRU (Least Recently Used) Cache :



`public int get(key)`

TimeComplexity : O(1)

`public void add(key, value)`

LRUCache size is fixed, if cache reaches the capacity,  
we would need to remove the “least recently used  
element”  
while adding the new element to the Cache.

`public LRUCache(int capacity) :`

LRUCache has the fixed capacity

`public void add(int key, int value) :`

Adds the element to LRUCache.

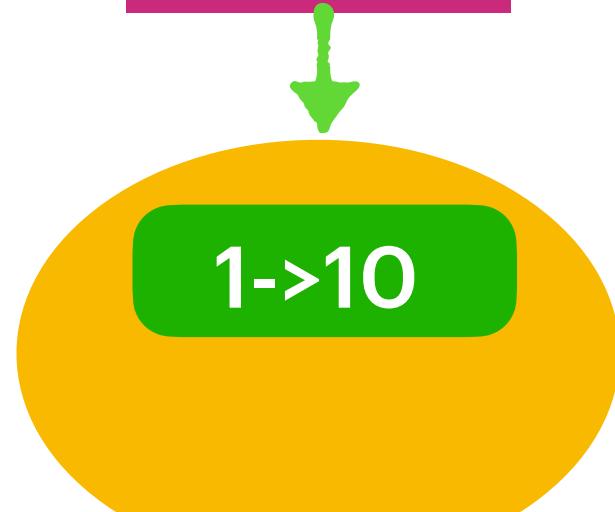
`public int get(int key) :`

Returns value if the key presents otherwise returns -1

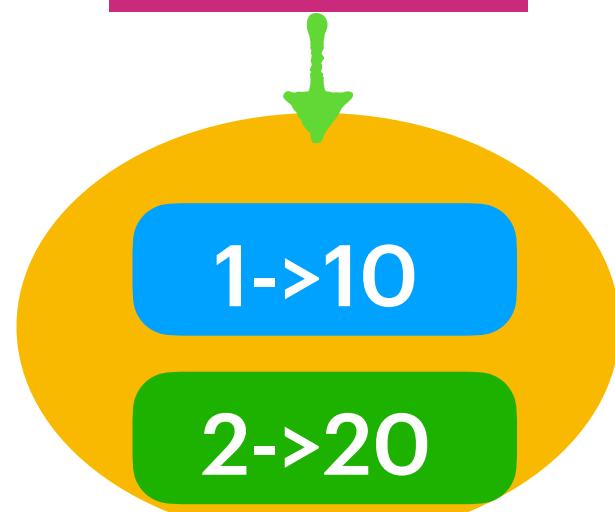
## LRUCache(capacity: 2)



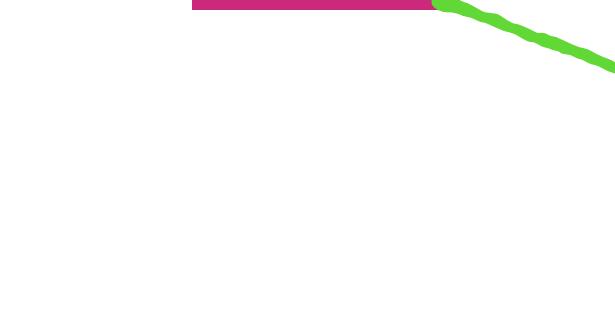
**add(1,10)**



**add(2,20)**



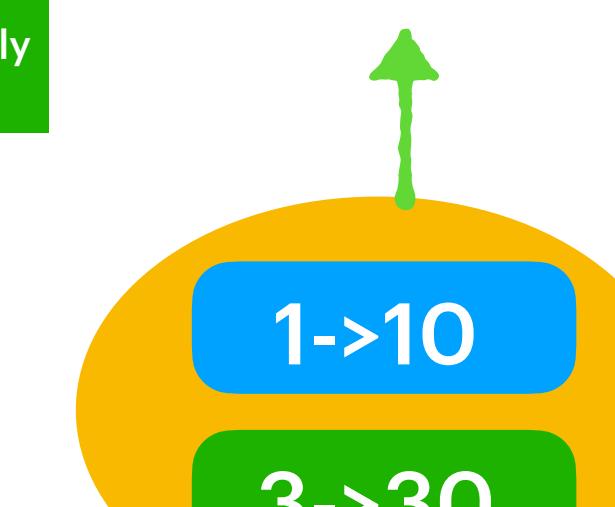
**get(1)**



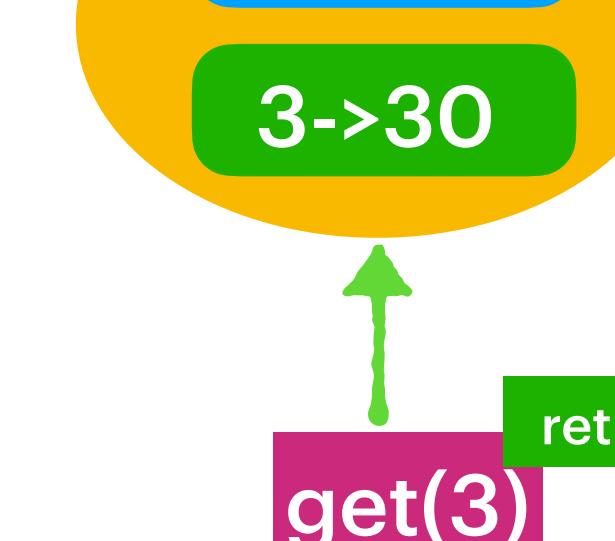
## Case 1 : Seeing all combinations

Removes 1->10 from cache because Which is the least recently used element in cache.

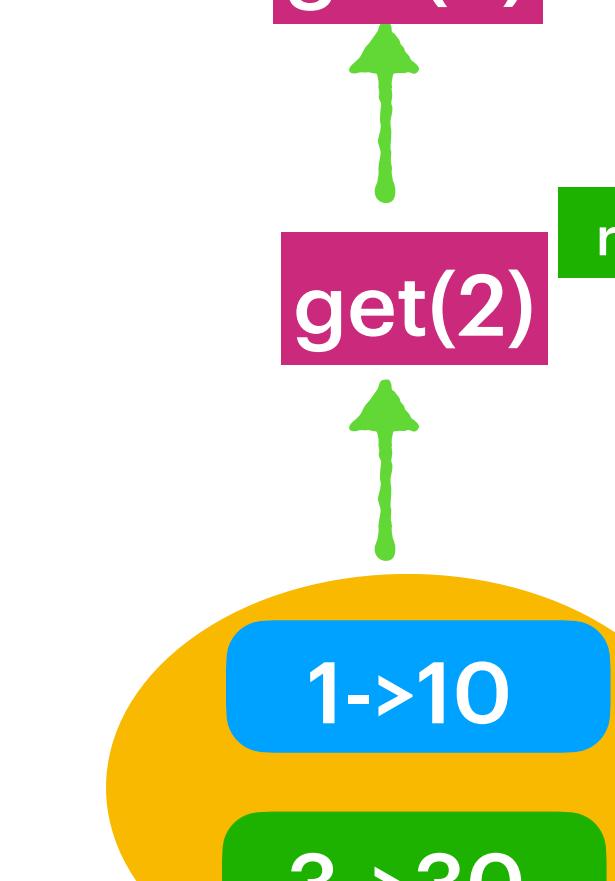
**add(4,40)**



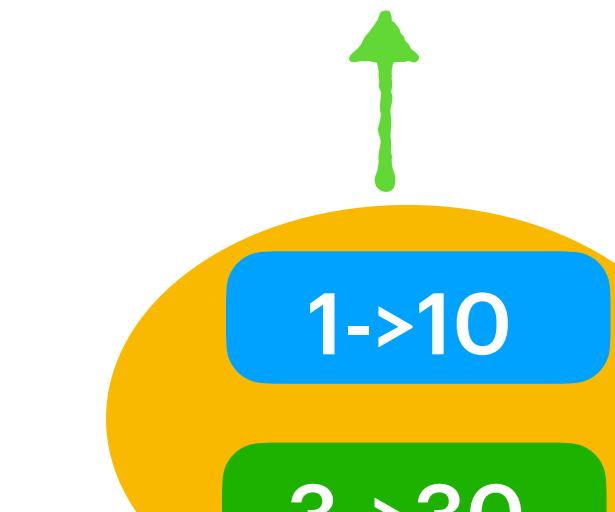
**get(1)**



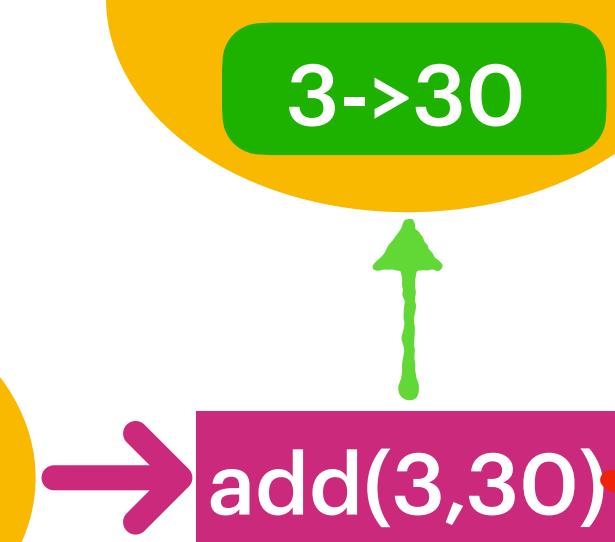
**get(3)**



**get(2)**



**get(4)**



**add(3,30)**



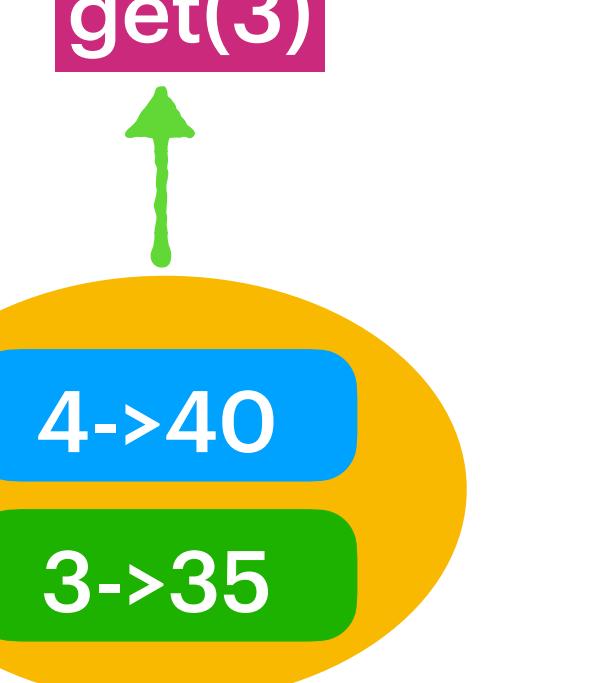
Represents Most Recently used element

Represents Least Recently used element

Represents Most Recently used element

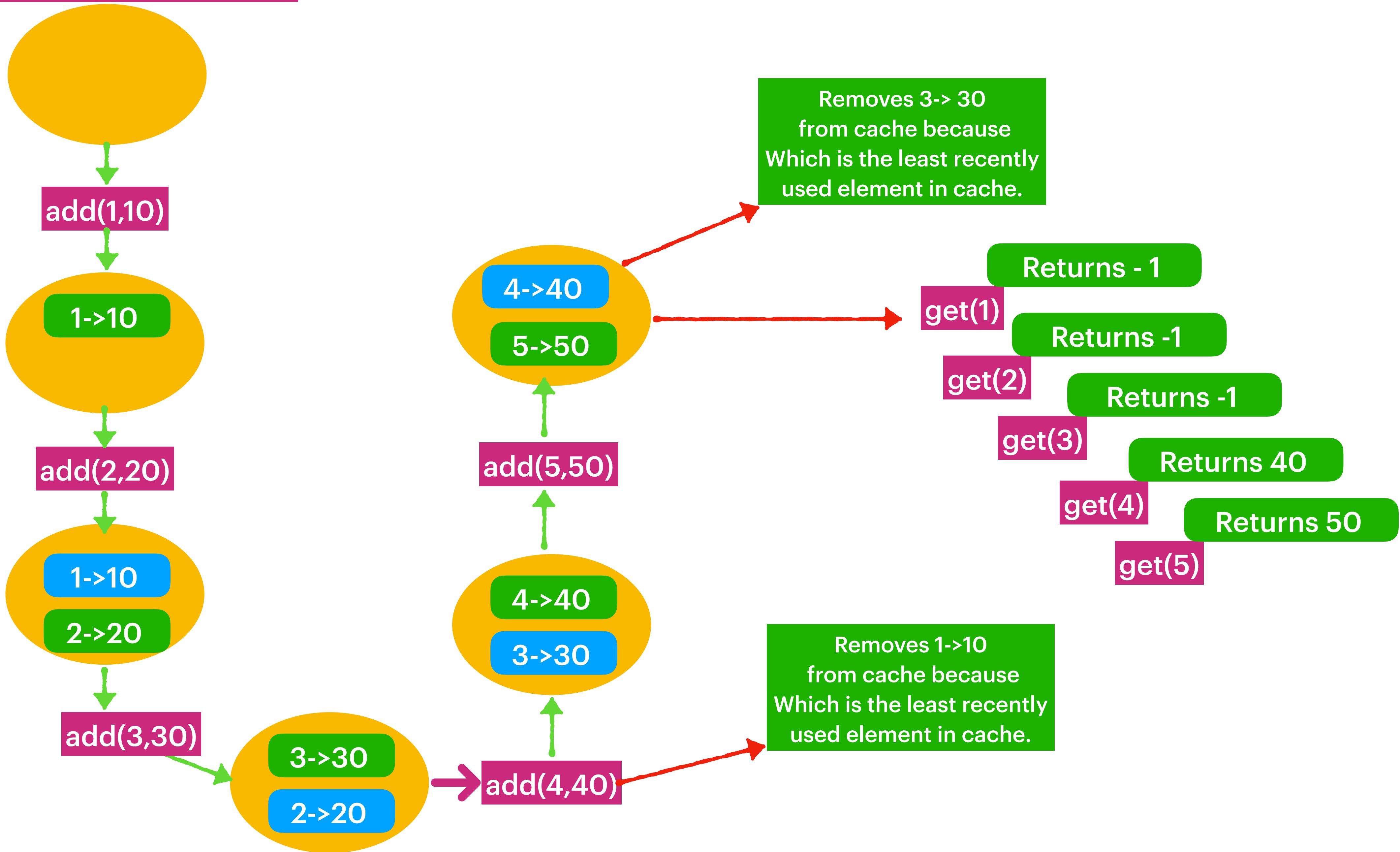
Represents Least Recently used element

**get(3)**



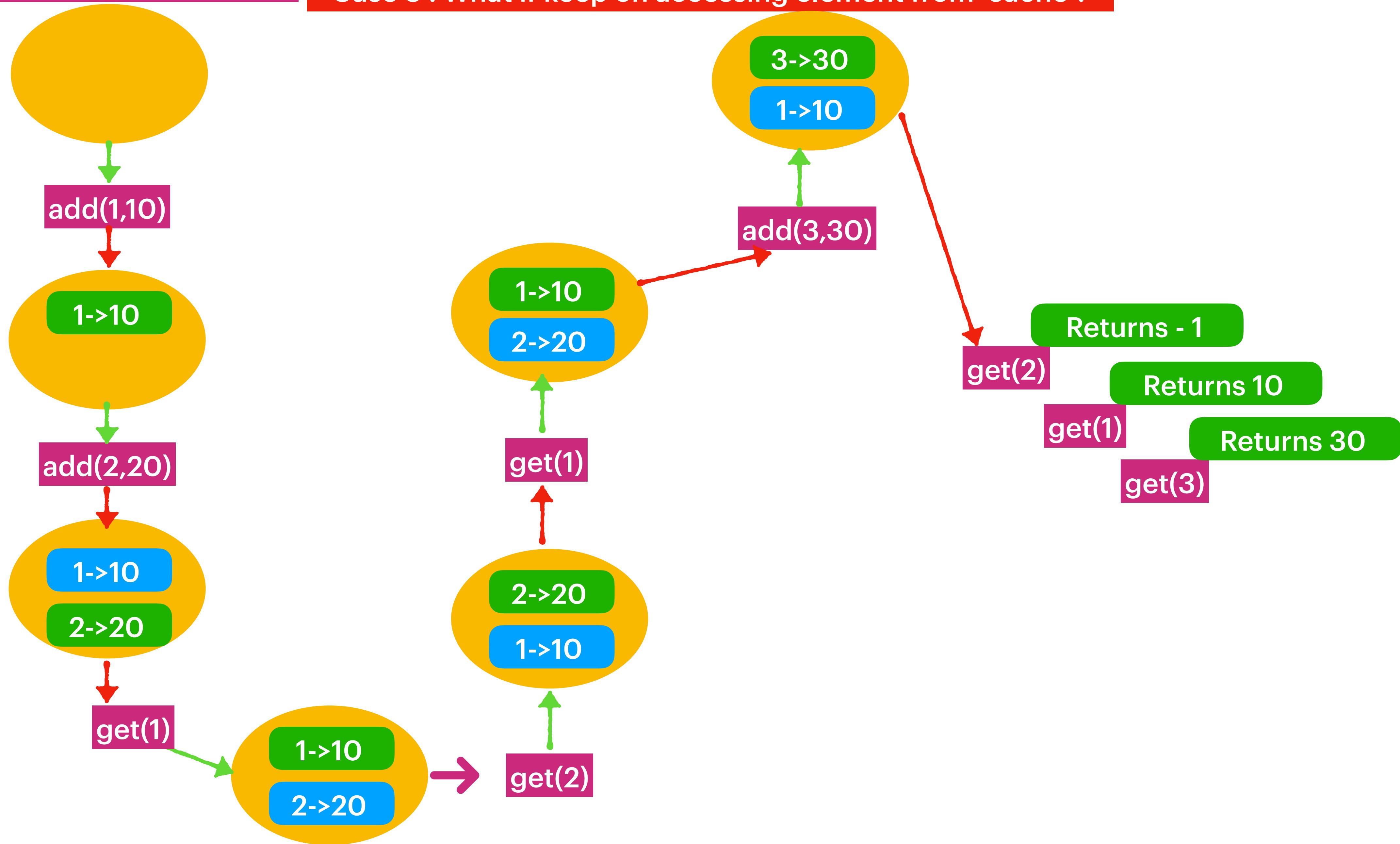
LRUCache(capacity: 2)

Case 2: What if keep on adding to cache ?



LRUCache(capacity: 2)

Case 3 : What if keep on accessing element from cache ?



## Double Linked List



Double Linked List has the reference of nextNode and its previous Node. So that we can traverse both in forward and reverse directions. Double Linked List simply fees the insert & delete operations.



```
class DLLNode {  
    int key,  
    int value,  
    Node next;  
    Node prev;  
}
```

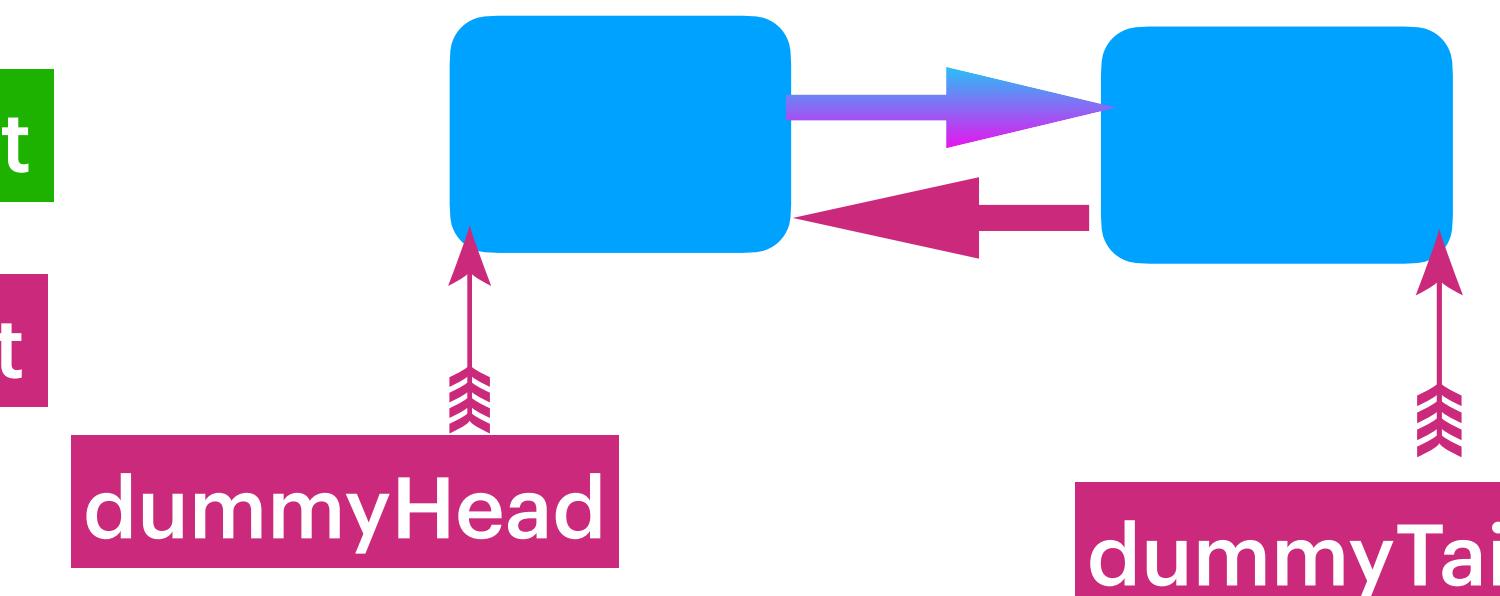
To avoid null checks maintain , take head & tail dummy nodes !!!

MRU : Most Recently Used Element

LRU : Least Recently Used Element

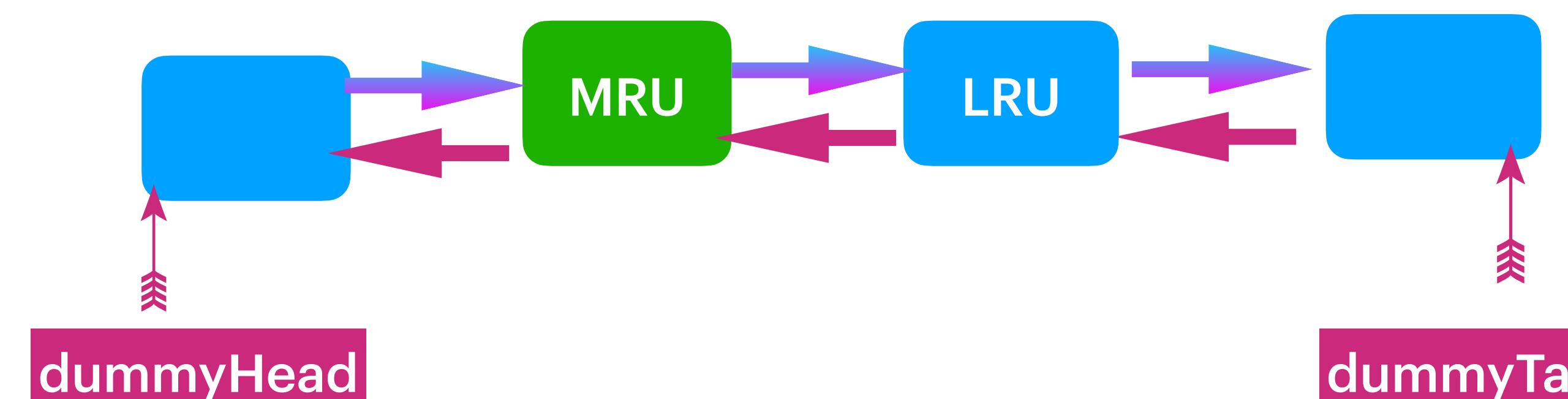
While  
add(key, value)  
(OR)  
get(key)

Add the element right  
after the dummyHead.



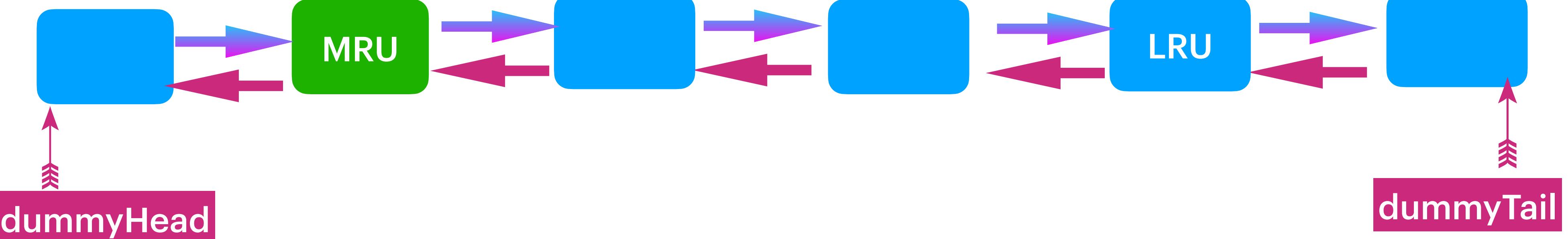
To maintain O(1) TimeComplexity  
Use Map<key,DLLNode> with DoubleLinkedList  
for add and get

Always add new node to right after head !!!  
It represents most recently used element.

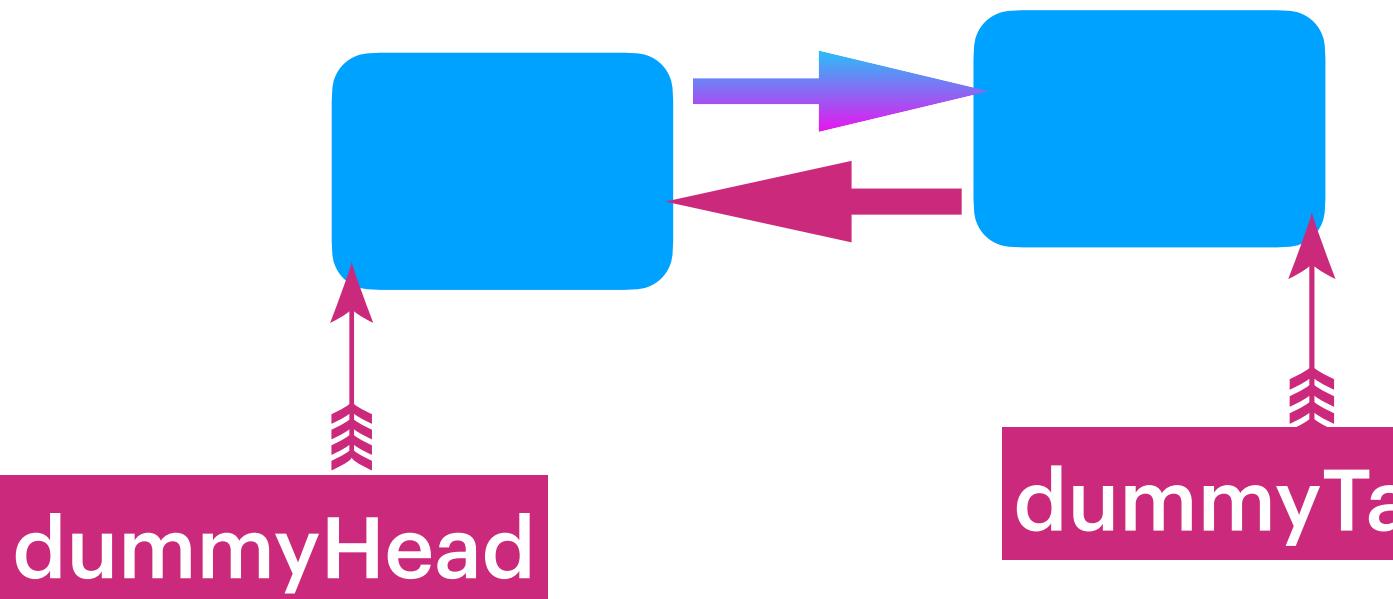


When the  
size > capacity

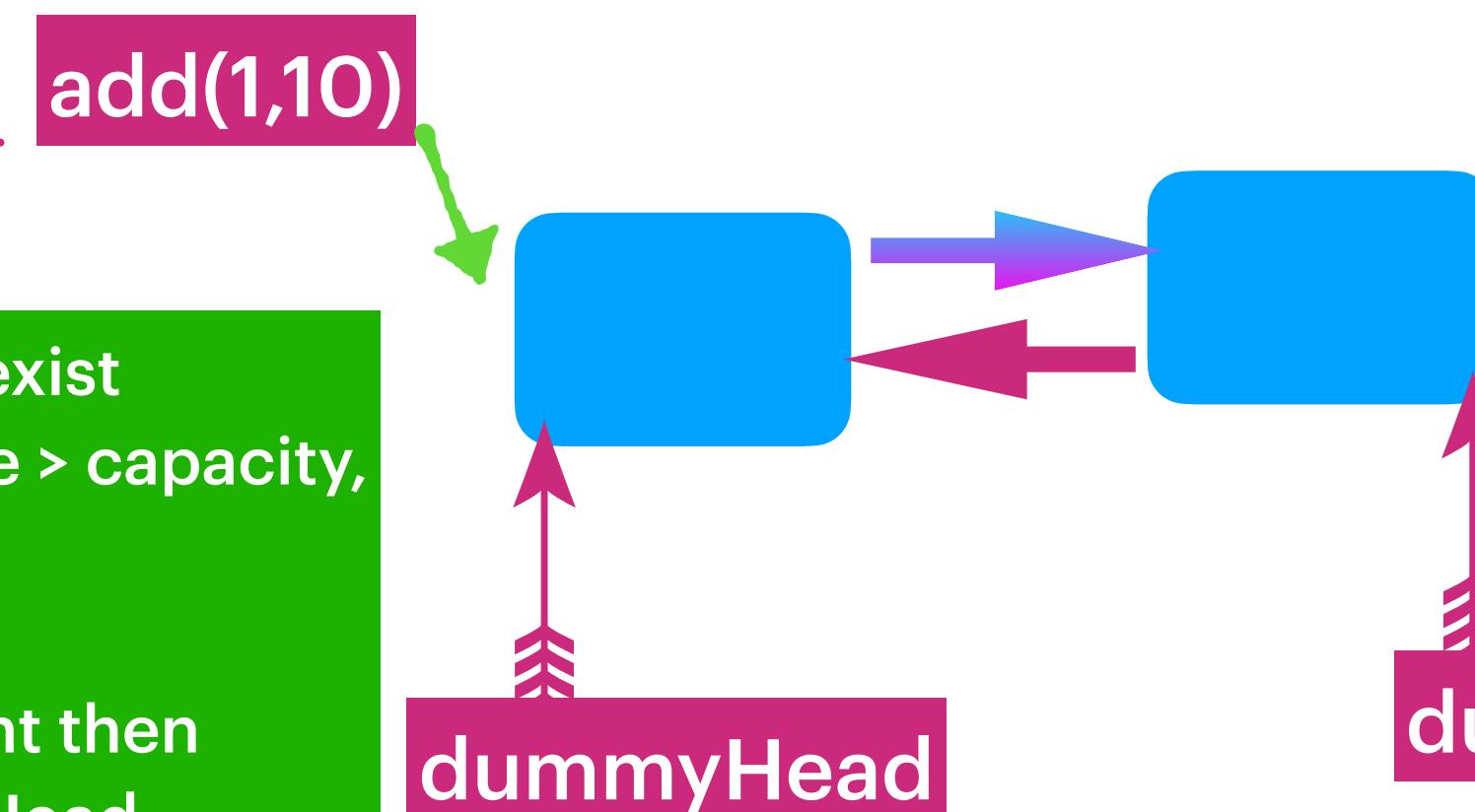
Remove  
LRU Element



To avoid null checks maintain , take head & tail dummy nodes !!!



To maintain O(1) TimeComplexity  
Use Map<key,DLLNode> with DoubleLinkedList  
for add and get



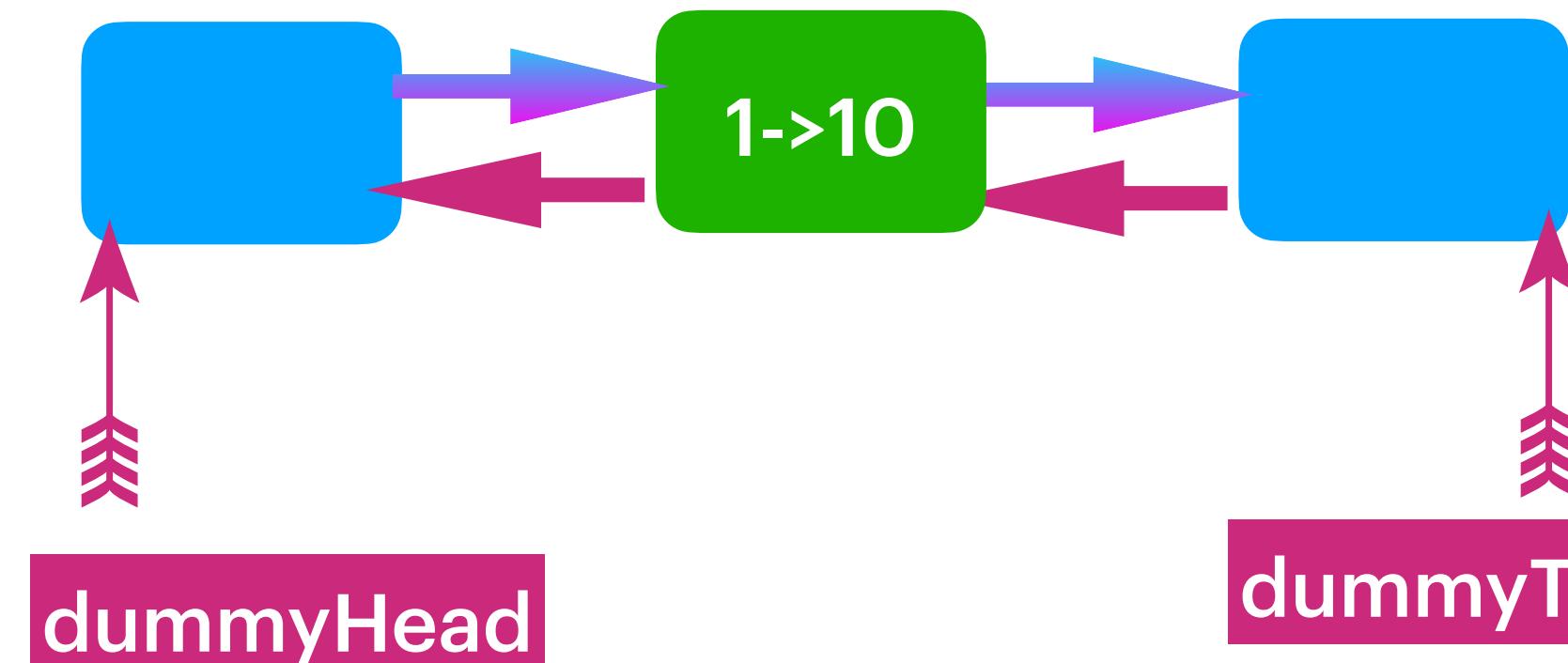
Always add new node to right after head !!!  
It represents most recently used element.

Case 1: If the node does exist  
addToHead & after add If the size > capacity,  
remove tail.prev.

Case 2: If the node is present then  
Update the value moveToHead.

LRUCache : Capacity(2)

Node(1->10) does not exist so add to the dummyHead

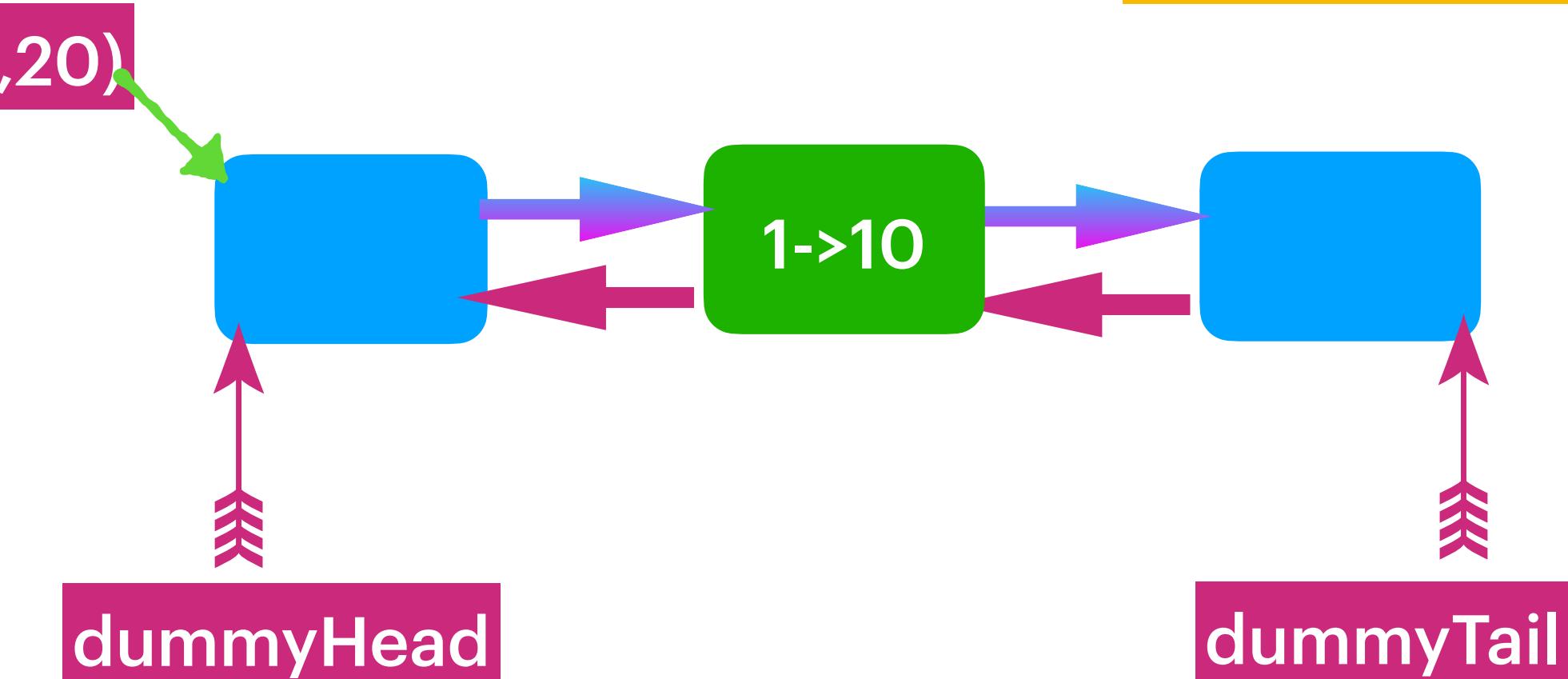


```
public void addToHead(DLLNode currentNode)
{
    DLLNode headNext = dummyHead.next;
    dummyHead.next = currentNode;
    headNext.prev = currentNode;
    currentNode.next = headNext;
    currentNode.prev = dummyHead;
    map.put(currentNode.key, currentNode);
    size++;
}
```

## LRUCache : Capacity(2)

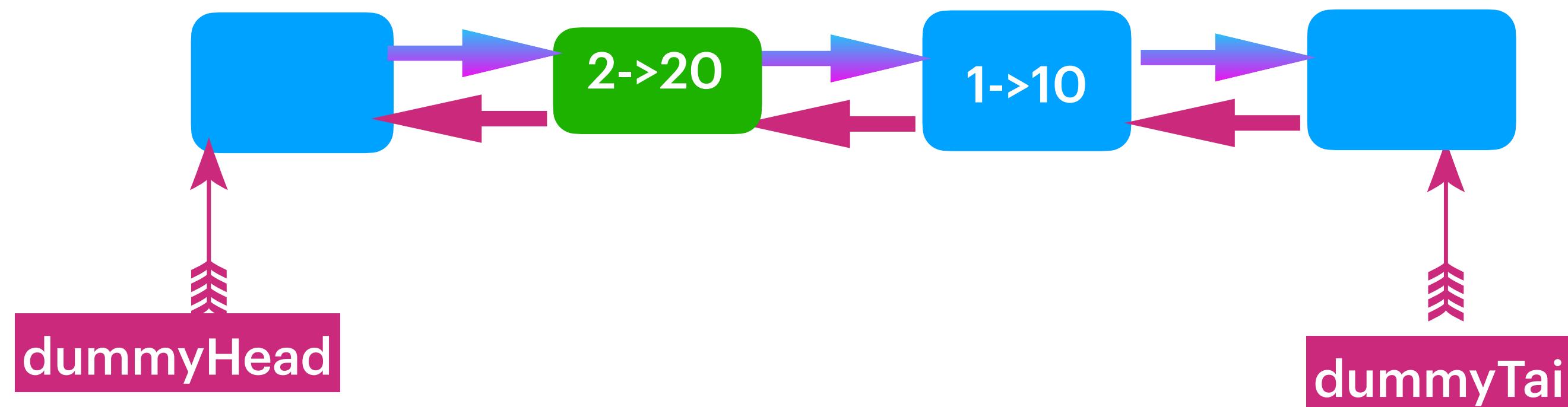
**Case 1:** If the node does exist addToHead & If the size > capacity, remove tail.prev.

**Case 2:** If the node is present then Update the value moveToHead.



```
public void addToHead(DLLNode currentNode)
{
    DLLNode headNext = dummyHead.next;
    dummyHead.next = currentNode;
    headNext.prev = currentNode;
    currentNode.next = headNext;
    currentNode.prev = dummyHead;
    map.put(currentNode.key, currentNode);
    size++;
}
```

Node(2->20) does not exist so add to the dummyHead



**get(1)**

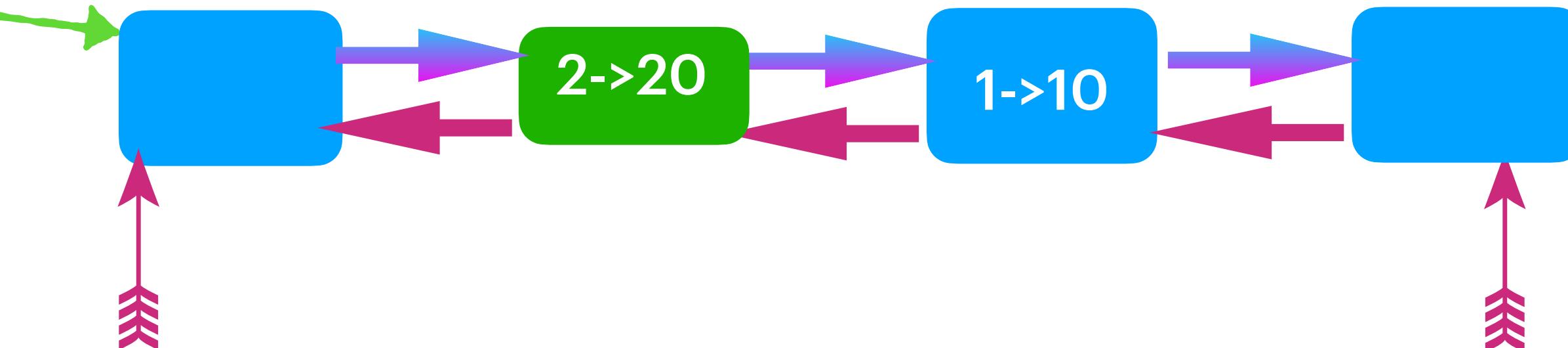
Move the accessed Node to the head

- 1) Remove currentNode From LinkedList.
- 2) Then Add the currentNode.

**dummyHead**

**dummyTail**

## LRUCache : Capacity(2)



```
public void removeNode(DLLNode currentNode)
{
    DLLNode prev = currentNode.prev;
    DLLNode next = currentNode.next;

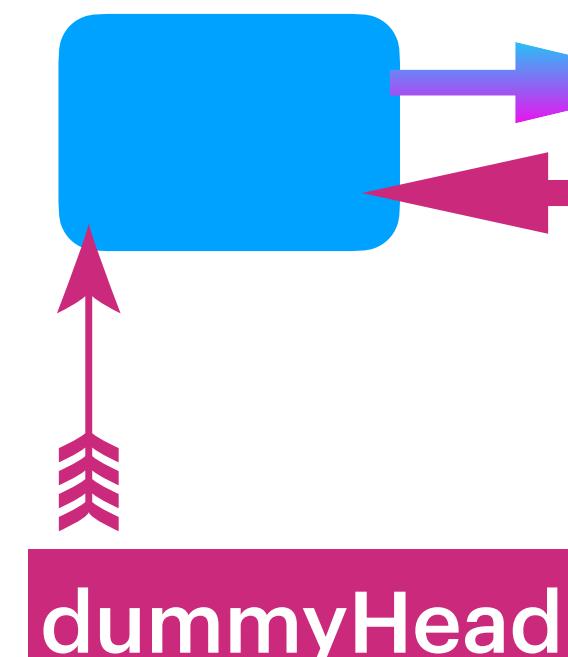
    prev.next = next;
    next.prev = prev;

    map.remove(currentNode.key);
    size--;
}
```

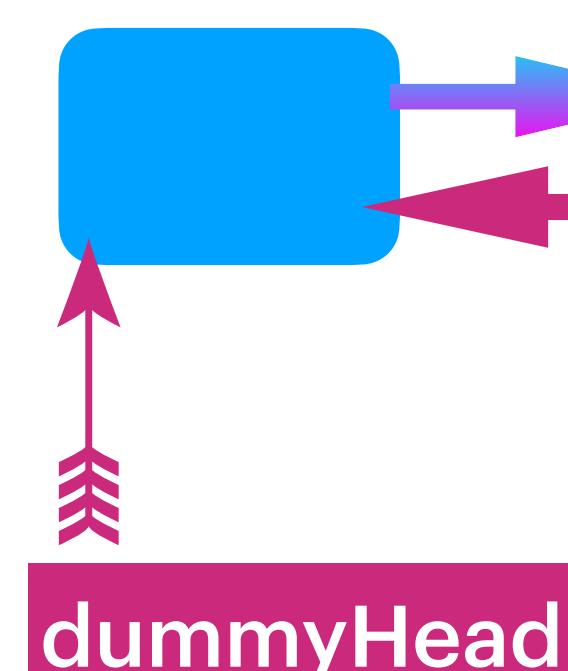
Remove currentNode(1)  
From LinkedList.

**dummyHead**

**dummyTail**



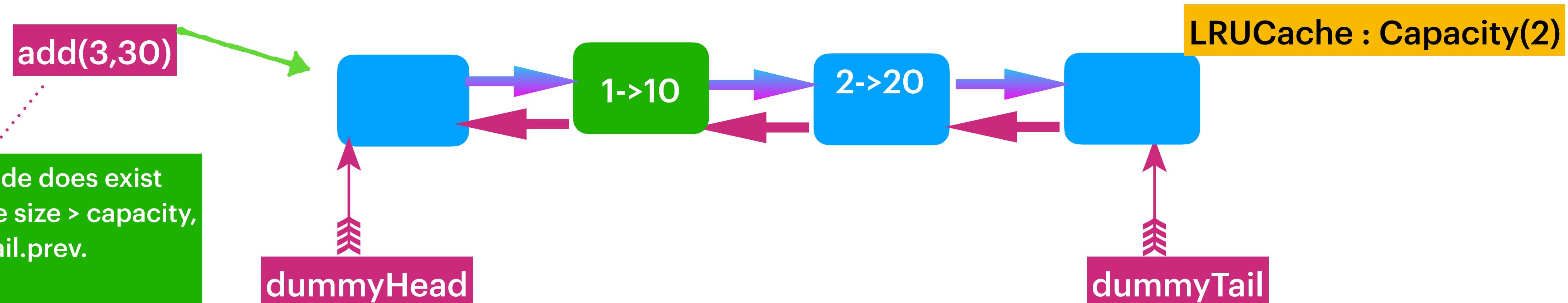
Then add currentNode(1)  
to the dummyHead.



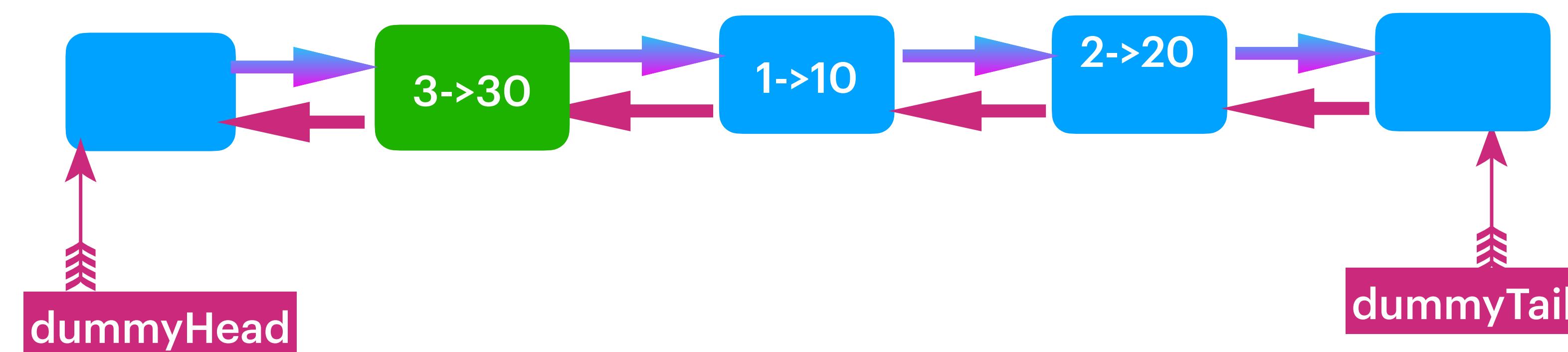
```
public void addNode(DLLNode currentNode)
{
    DLLNode headNext = dummyHead.next;
    dummyHead.next = currentNode;

    headNext.prev = currentNode;
    currentNode.next = headNext;
    currentNode.prev = dummyHead;

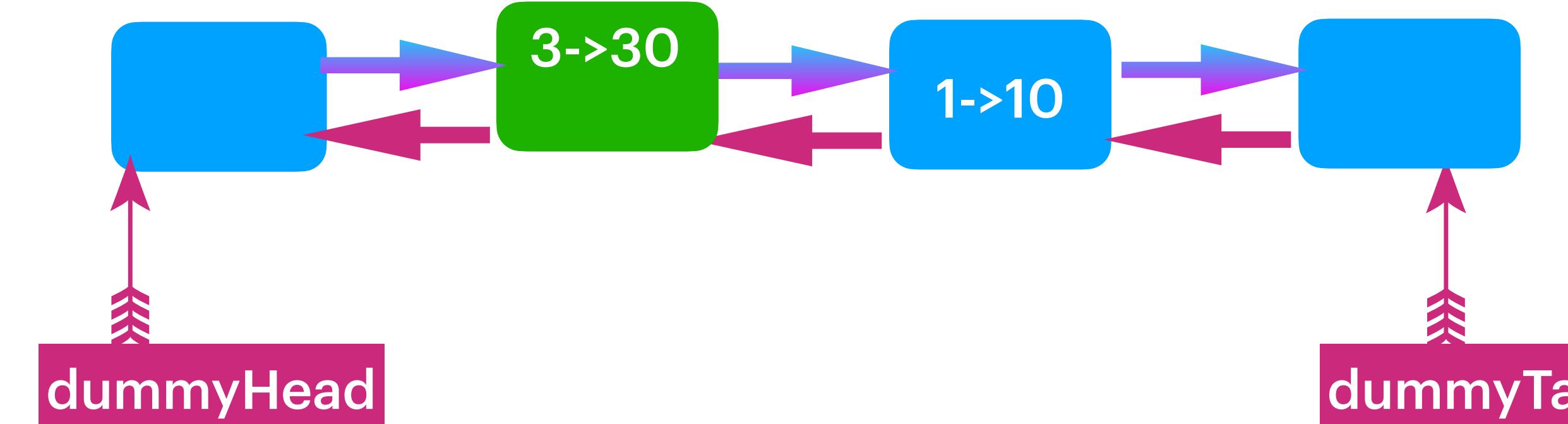
    map.put(currentNode.key, currentNode);
    size++;
}
```



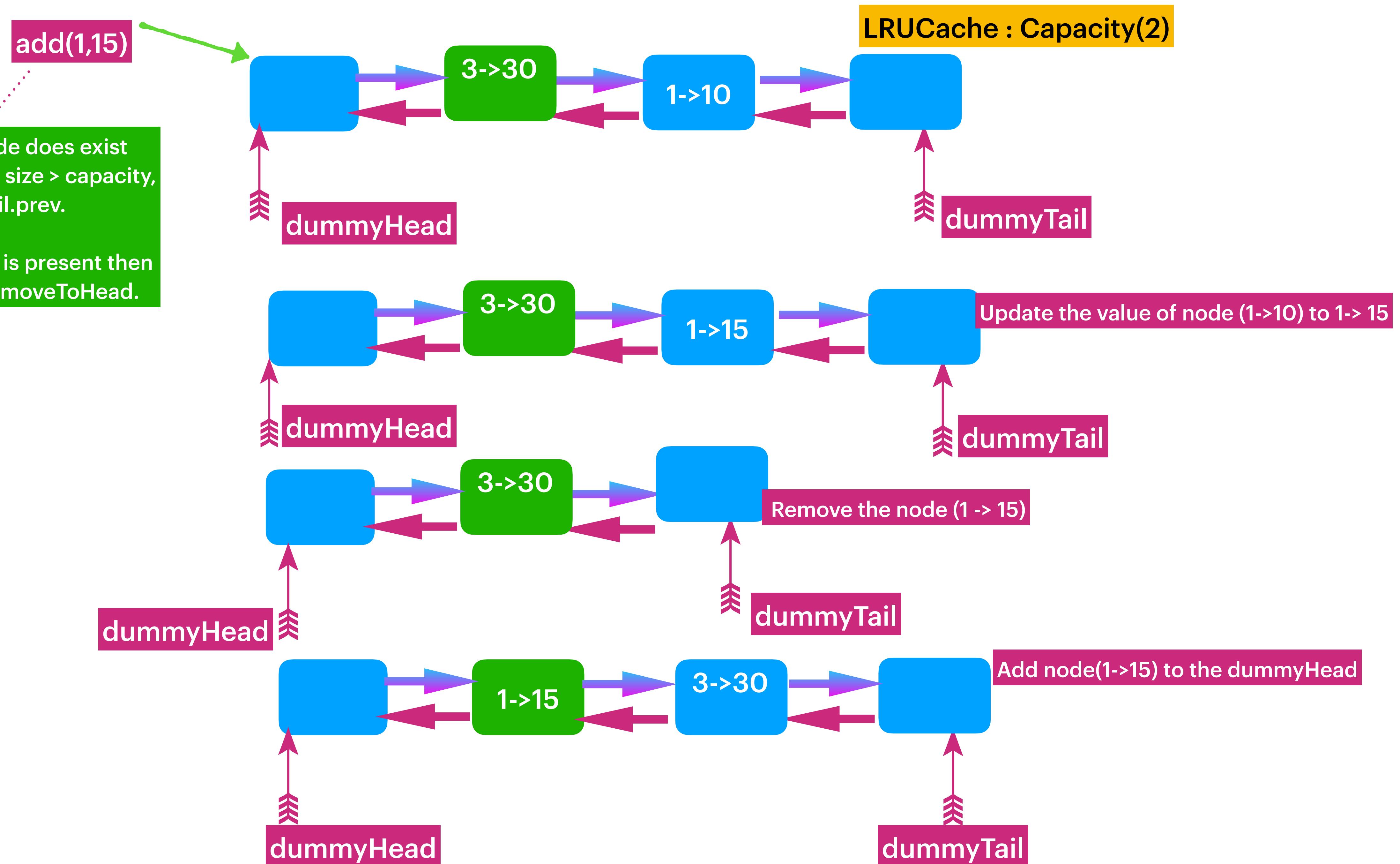
**Node(3->30) does not exist so add to Head**

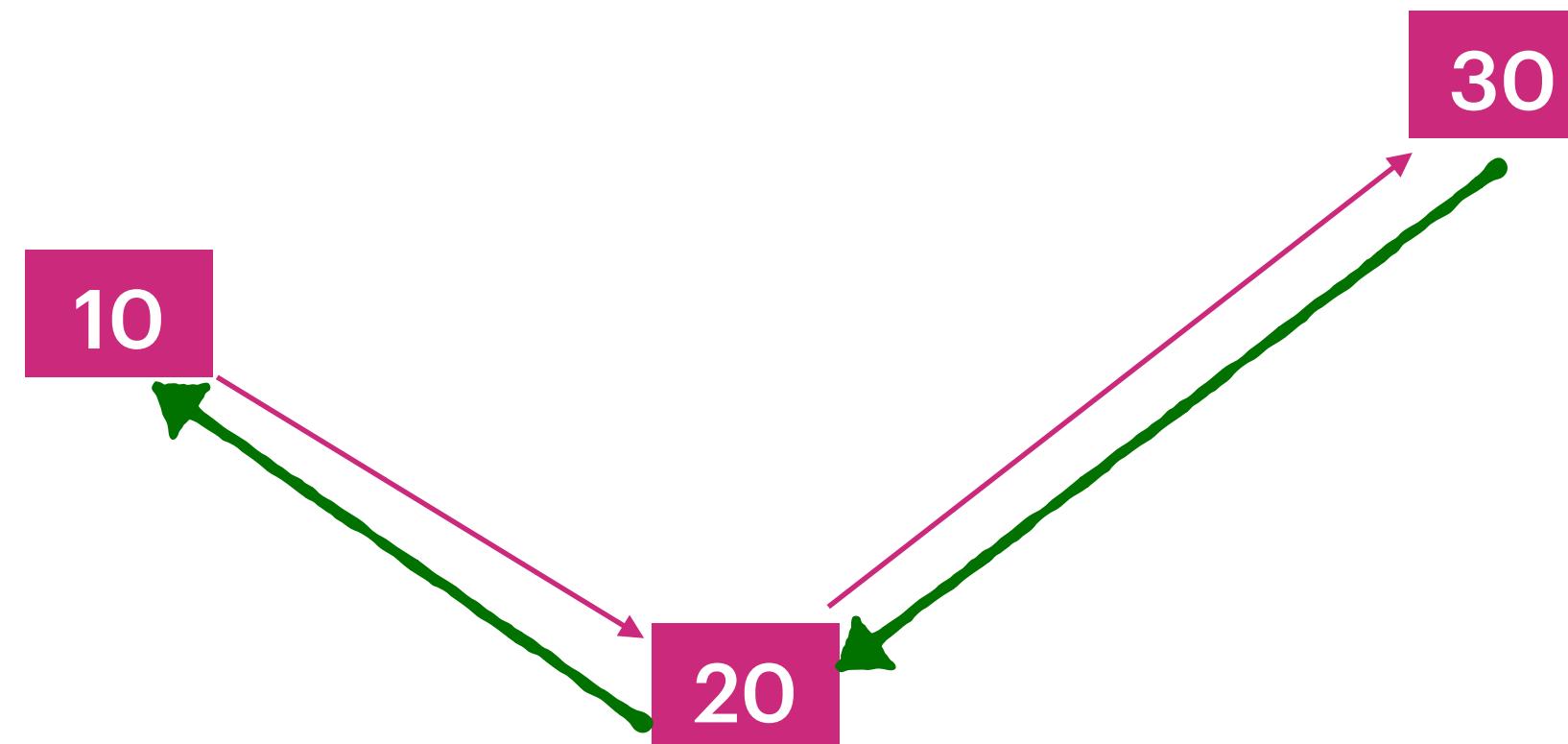


**Size > capacity so remove dummyTail.prev**



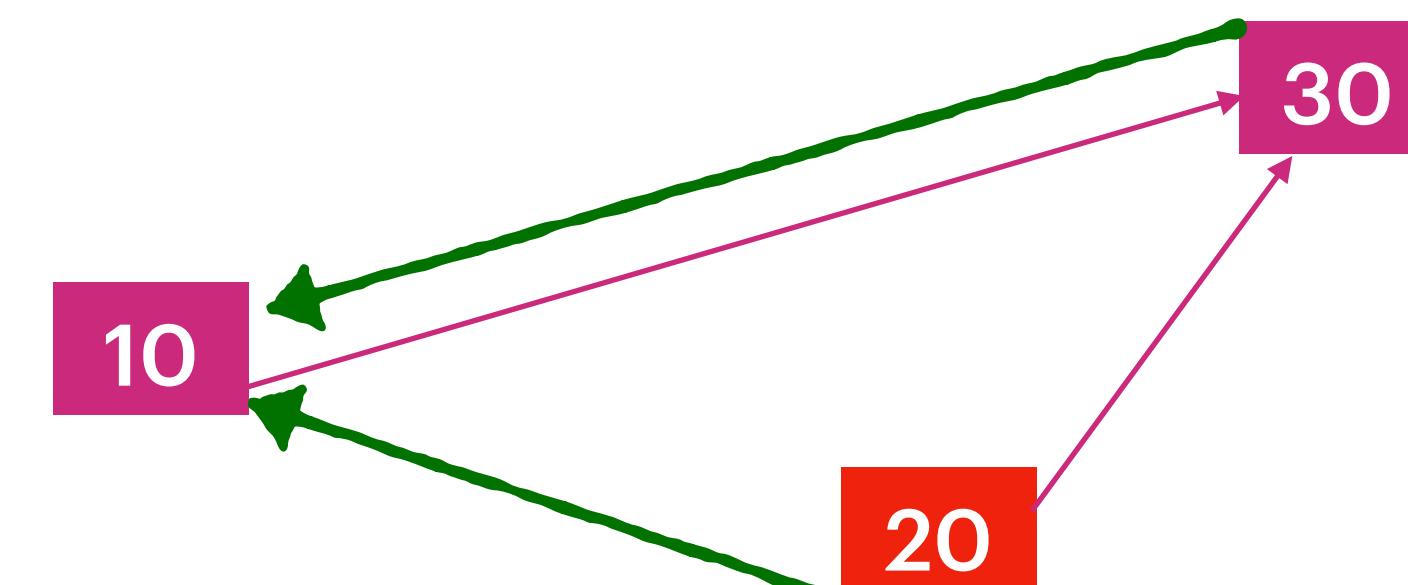
```
Remove dummyTail.prev
// dummyTail.prev always maps to LRU Element
public void removeTail()
{
    DLLNode tailPrev = dummyTail.prev;
    removeNode(tailPrev);
}
```





Remove(20)::

```
DLLNode prev = currentNode.prev;  
DLLNode next = currentNode.next;  
  
prev.next = next;  
next.prev = prev;
```



As node(20) is not referred by any reference  
eligible for Garbage Collection

## Design LFU (Least Frequently Used) Cache :



`public int get(key)`

TimeComplexity : O(1)

`public void add(key, value)`

LFUCache size is fixed, when the cache is full,  
we would need to remove the “Least Frequently Used ”(LFU)  
element.

There is a possibility that multiple elements could be accessed  
equally, in such case remove older LFU element.

`public LFUCache(int capacity) :`

LFUCache has the fixed capacity

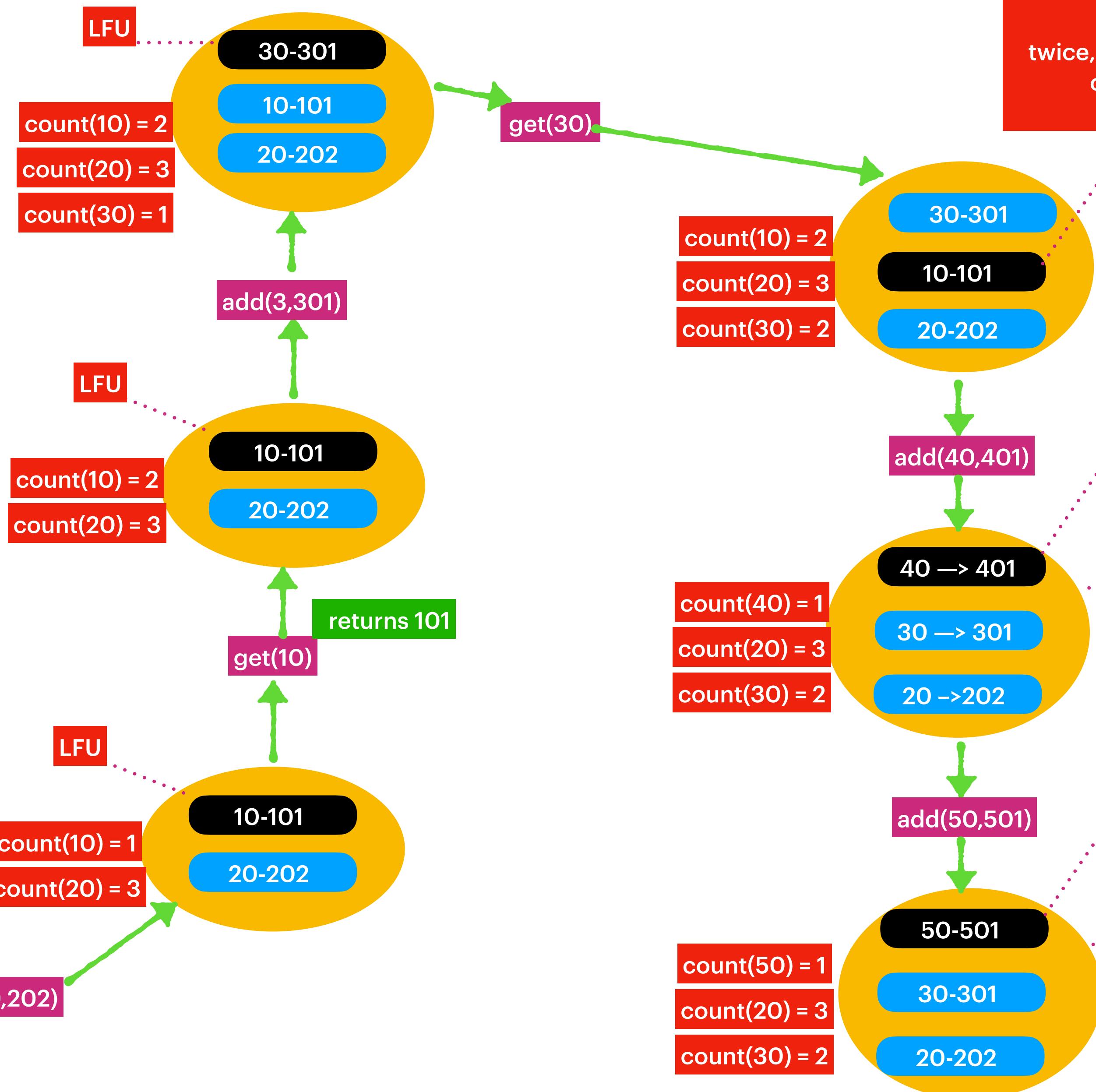
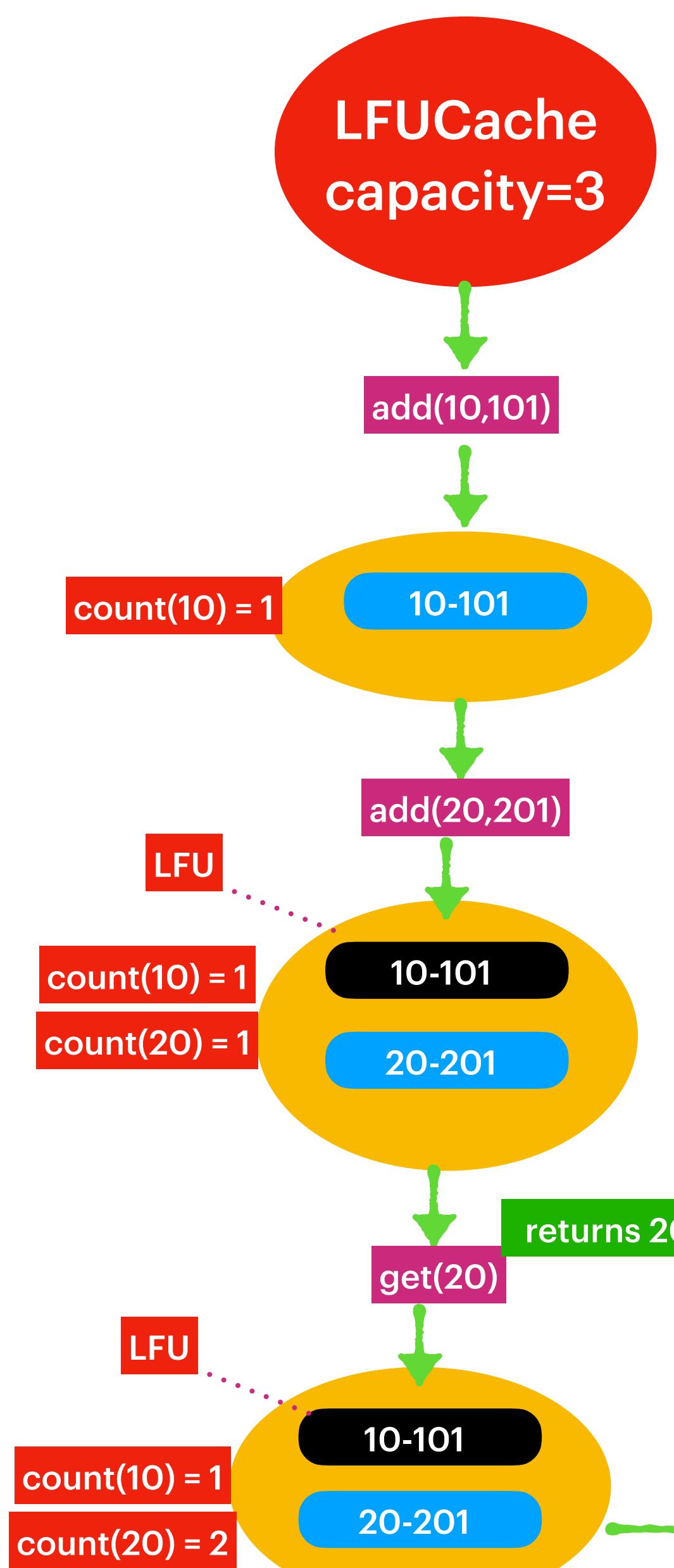
`public void add(int key, int value) :`

adds /updates the element to the LFUCache.  
If the cache is full then removes the LFU element  
then adds the new one.

`public int get(int key) :`

Returns value if the key presents otherwise returns -1

# LFUCache(capacity: 3)



LFU is node(10):  
Ofcourse 10,30 accessed twice, when the count is same we should consider, old element as LFU.  
Here node(10) is old element

By add of node(40) then the size > capacity so removed LFU element i.e node(10)

By add of node(50) then the size > capacity so removed LFU element i.e node(40)

Finally Output of following get Calls

get(10)	returns -1
get(20)	returns 301
get(30)	returns 202
get(40)	returns -1
get(50)	returns 501

## Algorithm For LFU Cache

We would need to remove Least Frequently Used (LFU) element :  
Constraints : get(key), add(key, value) should be done in O(1) time.



Maintain two Maps

1. ElementsMap => Here key is input-key, value is DLLNode:  
 $\text{Map} < \text{key}, \text{DLLNode} > \text{elementsMap}$

2. CounterMap=> Here key is the counter and value would be LRUCache.

$\text{Map} < \text{counter}, \text{LRUCache} > \text{elementsMap}$

Why LRUCache?

When multipleNodes accessed in equal time then all the nodes have same counter.  
We would need to remove older node so that LRUCache can delete older element in O(1) time.

So in counterMap each counterKey represents on LRUCache.

3. Use auxiliary space / temporary variable which maintains minFrequencyCounter value.

Why auxiliary space / temporary variable?

When the cache is filled we can identity the LFU element using minFrequencyCounter  
then can be removed

From both CounterMap & ElementsMap in O(1) time.

```
Map<Integer, DLLNode> elementsMap = new HashMap<>();  
Map<Integer, LRUcache> counterMap = new HashMap<>();  
Int minFrequencyCounter = Integer.MAX_VALUE;
```

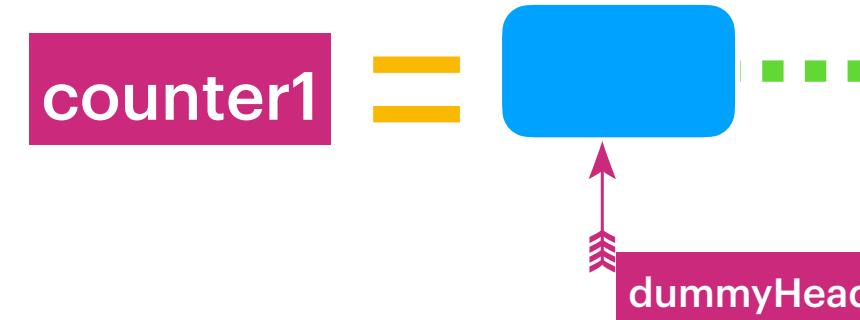
### class DLLNode

```
{  
    private int key;  
    private int value;  
    private int counter;  
    private DLLNode next;  
    private DLLNode prev;  
}
```

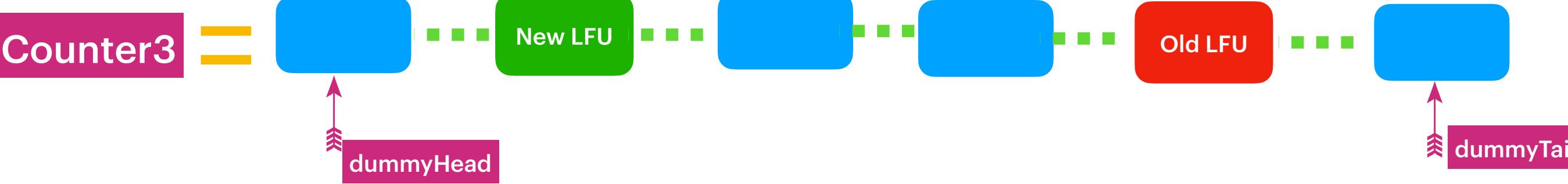
In CounterMap For each  
Counter key : we use LRUcache  
So that removing of LFU element  
would be done in constant time O(1)

counterMap

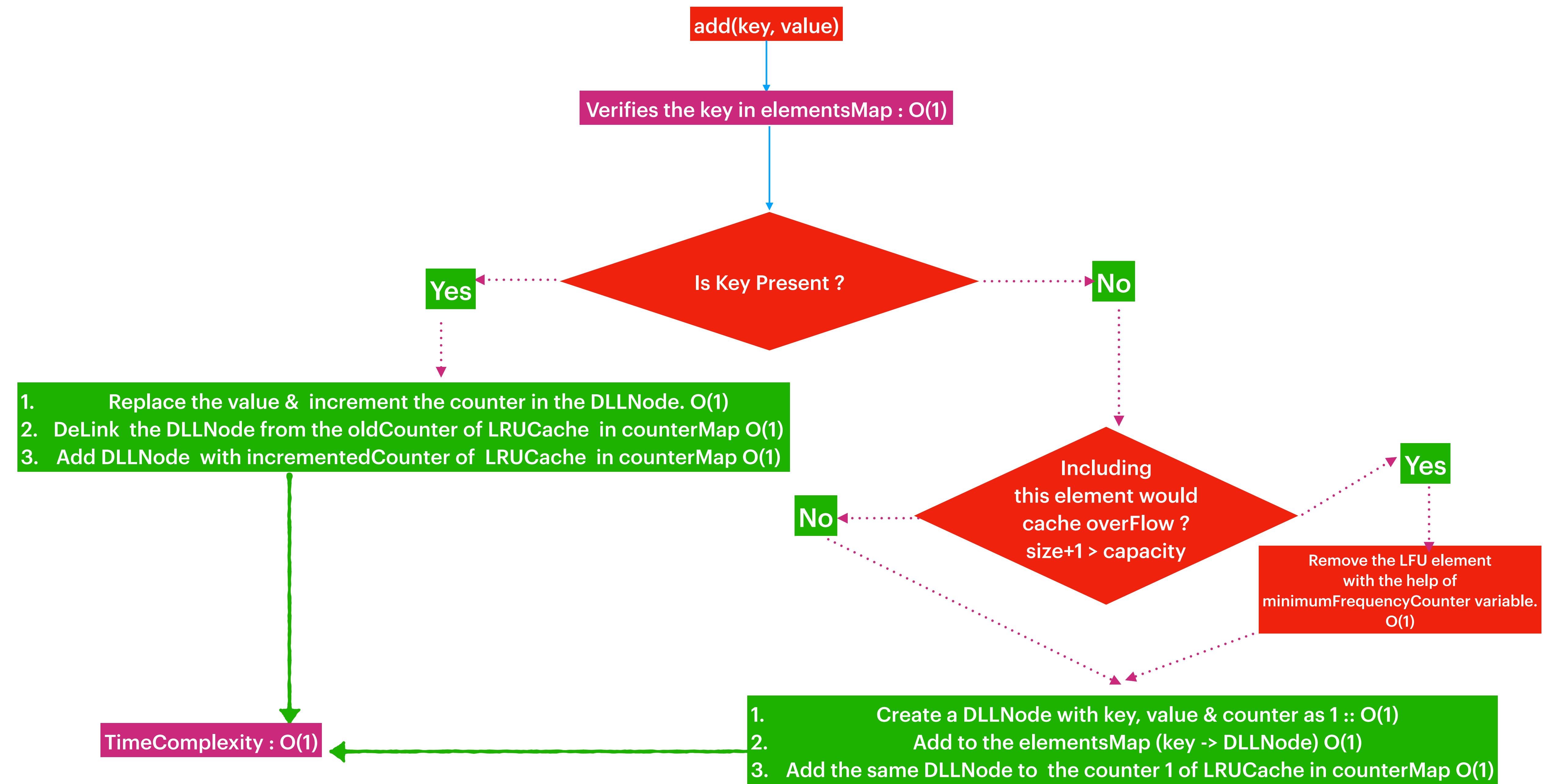
[

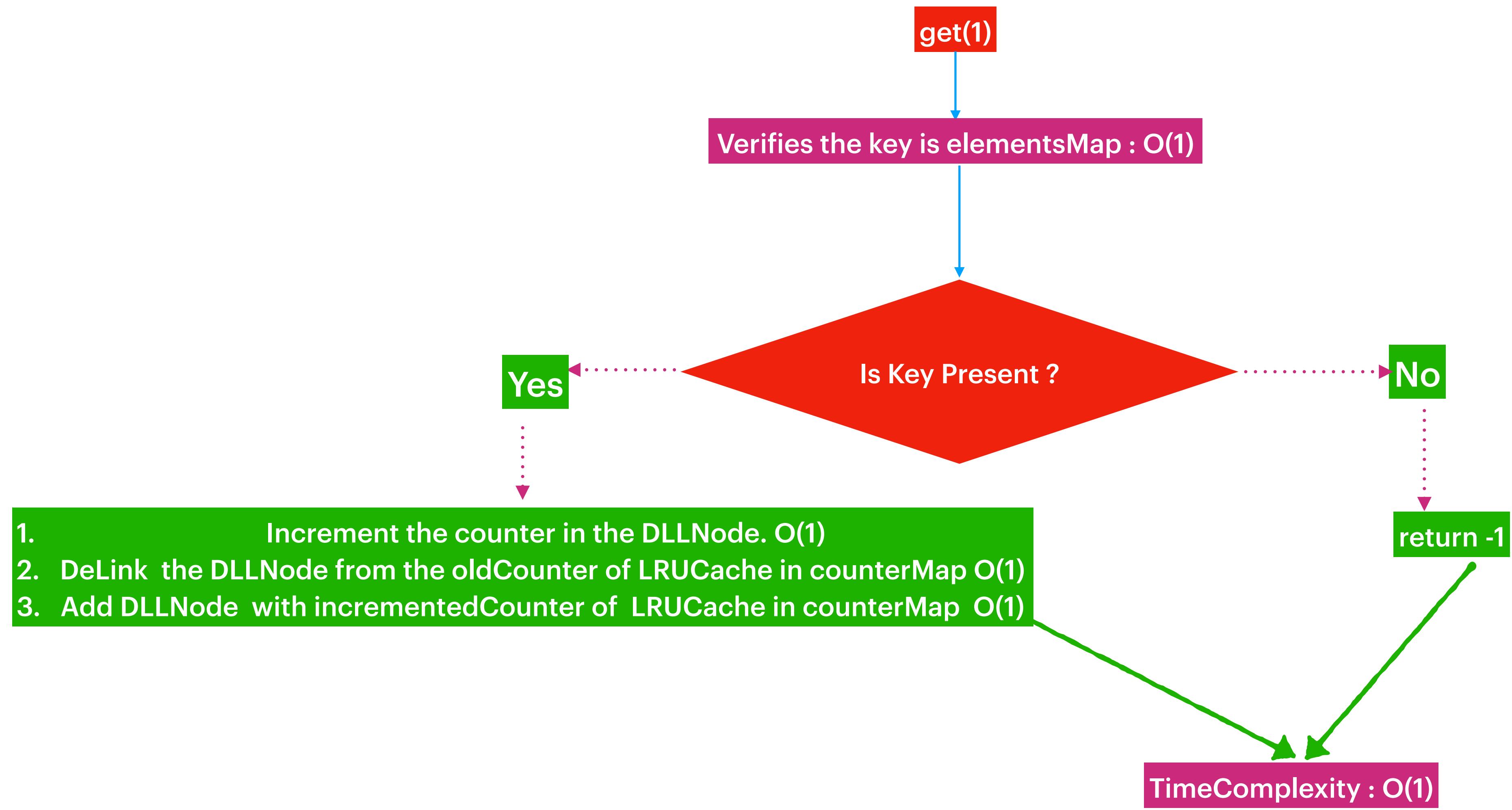


Here LRUcache maintains a order of LFU elements  
for each counter from new generation  
to old generation.



]





Step1: Add node(10) to the elementMap key : 10 & initial counterValue =1,  
Step2: Link node(10) in counterMap for the counterKey=1

## LFUCache(capacity: 3)

minFreqCounterKey helps us to fetch  
associated LFU Nodes.  
in counterMap.

LFU Nodes???

Yes its LFU Nodes, there is a possibility that  
with the same counter, multiple nodes exists.

We can use minFreqCounterKey , to delete LFU node in  
O(1) time when the the cache is filled.

key(10) is not presented in elementsMap.  
elementsMap[ ]  
add(10,101)

elementsMap:  
[  
10 -> node(10, 101, c1)  
Step1  
]  
Step1

Link the key(10) with the counterKey = 1

minFreqCounterKey = 1

counterMap:

[  
1 ->[dummyHead ->node(10)-> dummyTail]  
]  
Step2  
Step2

key(20) is not presented in elementsMap.  
elementsMap[10 -> node(10) ]  
add(20,201)

elementsMap:  
[  
10 -> node(10, 10v, c1)  
20 -> node(20, 20v, c1)  
Step1  
]  
Step1

Link the key(20) with the counterKey = 1

minFreqCounterKey = 1

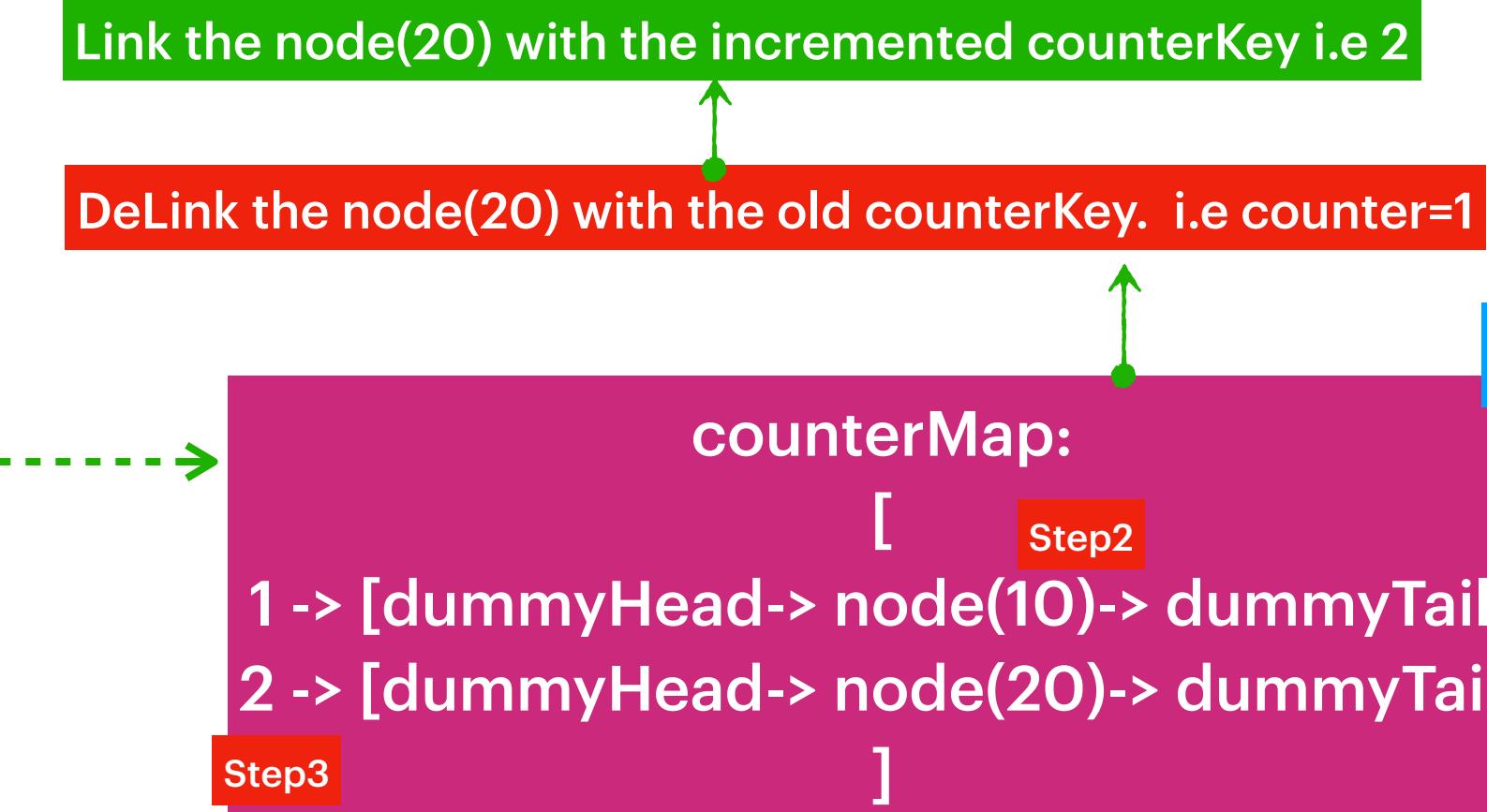
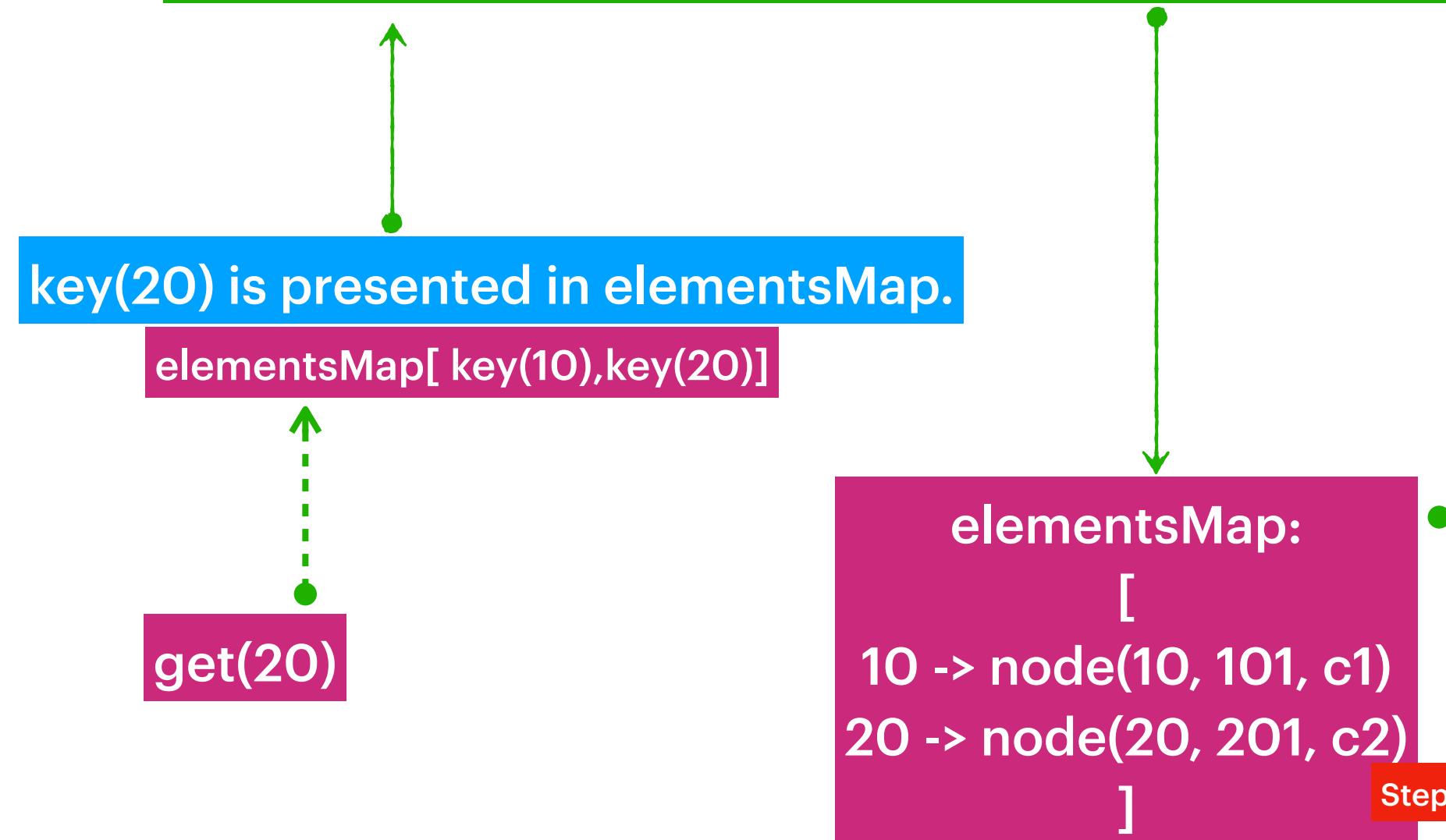
counterMap:

[  
1 -> [dummyHead ->node(20) -> node(10)-> dummyTail]  
]  
Step2  
Step2

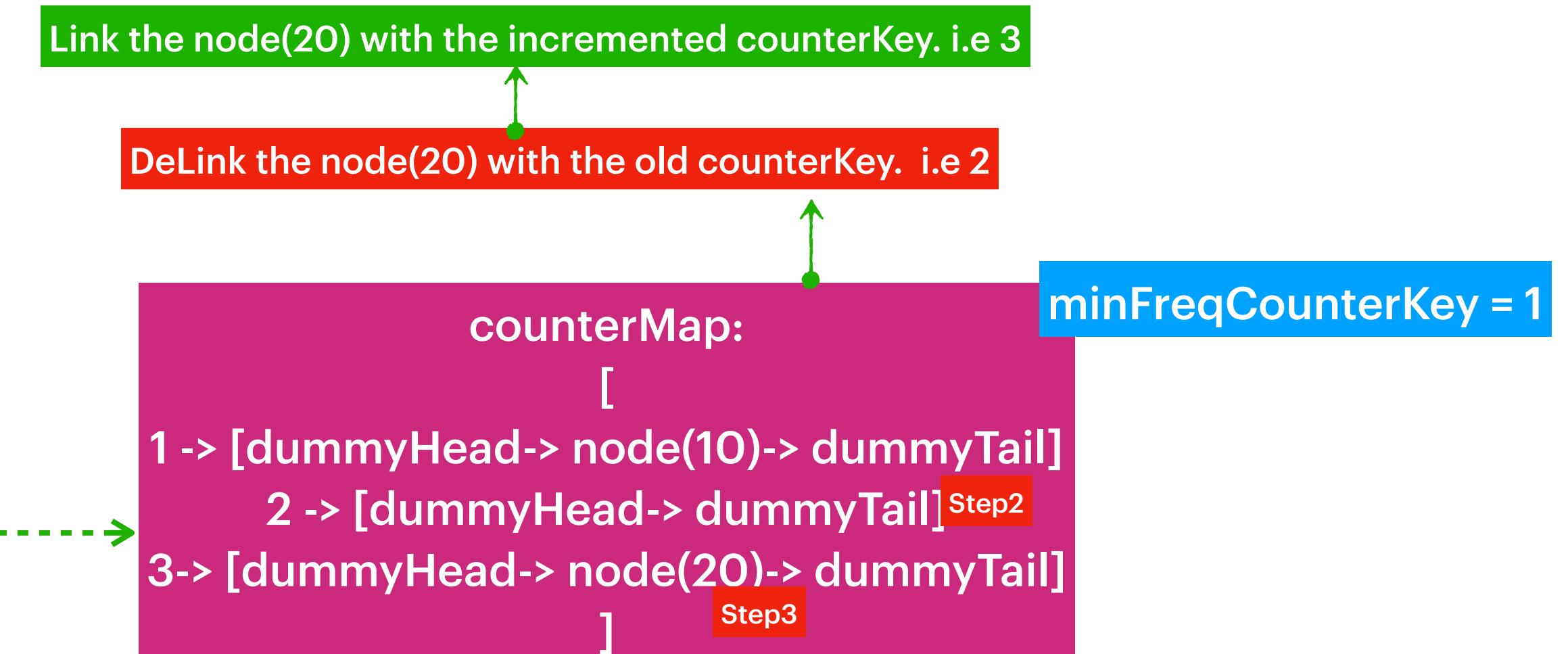
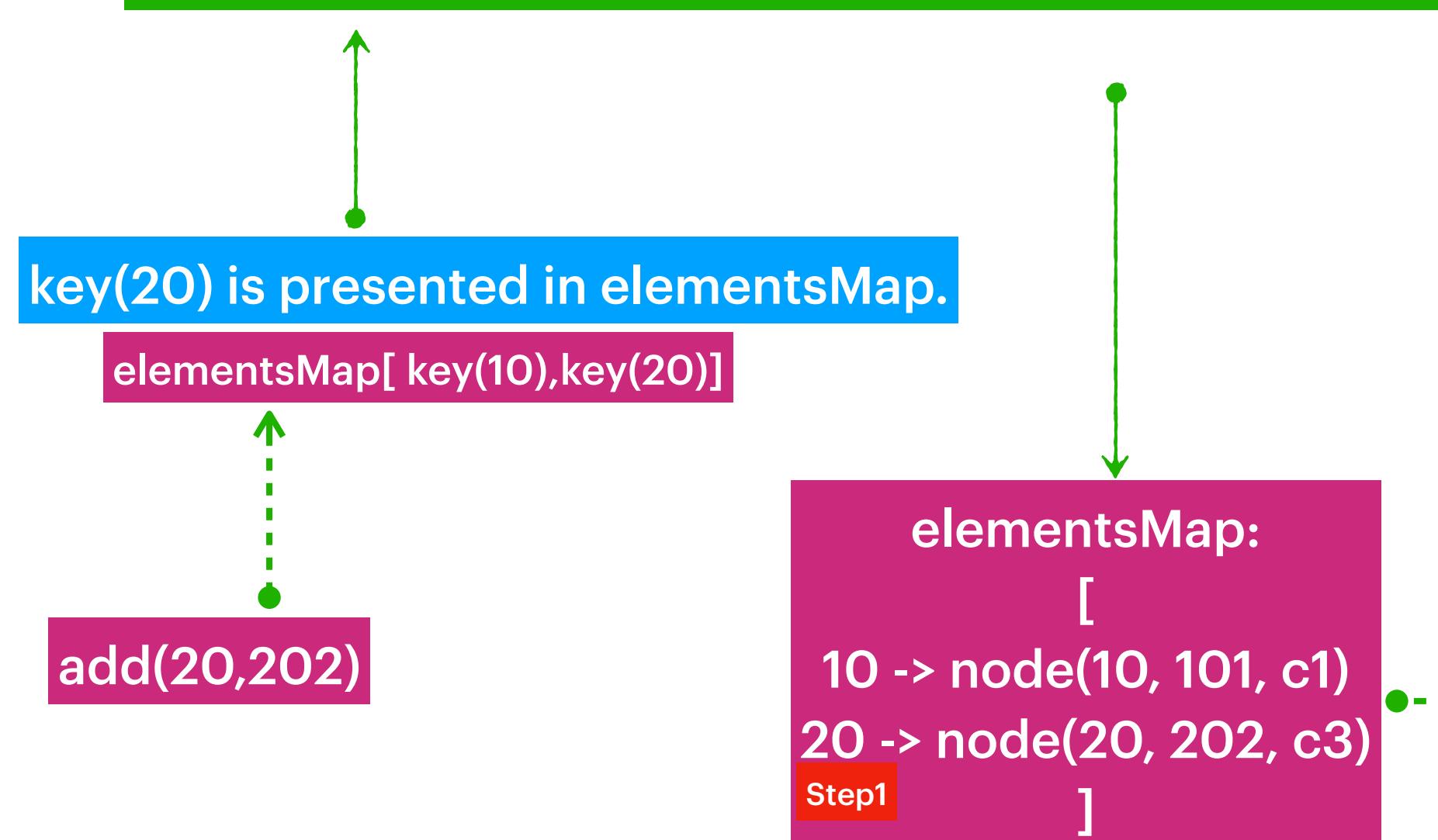
Adding new node to the right of dummyHead  
is the Concept of LRUCache so that we can  
consider dummyTail.prev as old LFU element.

## LFUCache(capacity: 3)

Step1: increment the counter for node(20).  
 Step2: In counterMap, DeLink the node(20) with the old counterKey i.e 1  
 Step3: Link the node(20) with the incremented counterKey i.e 2

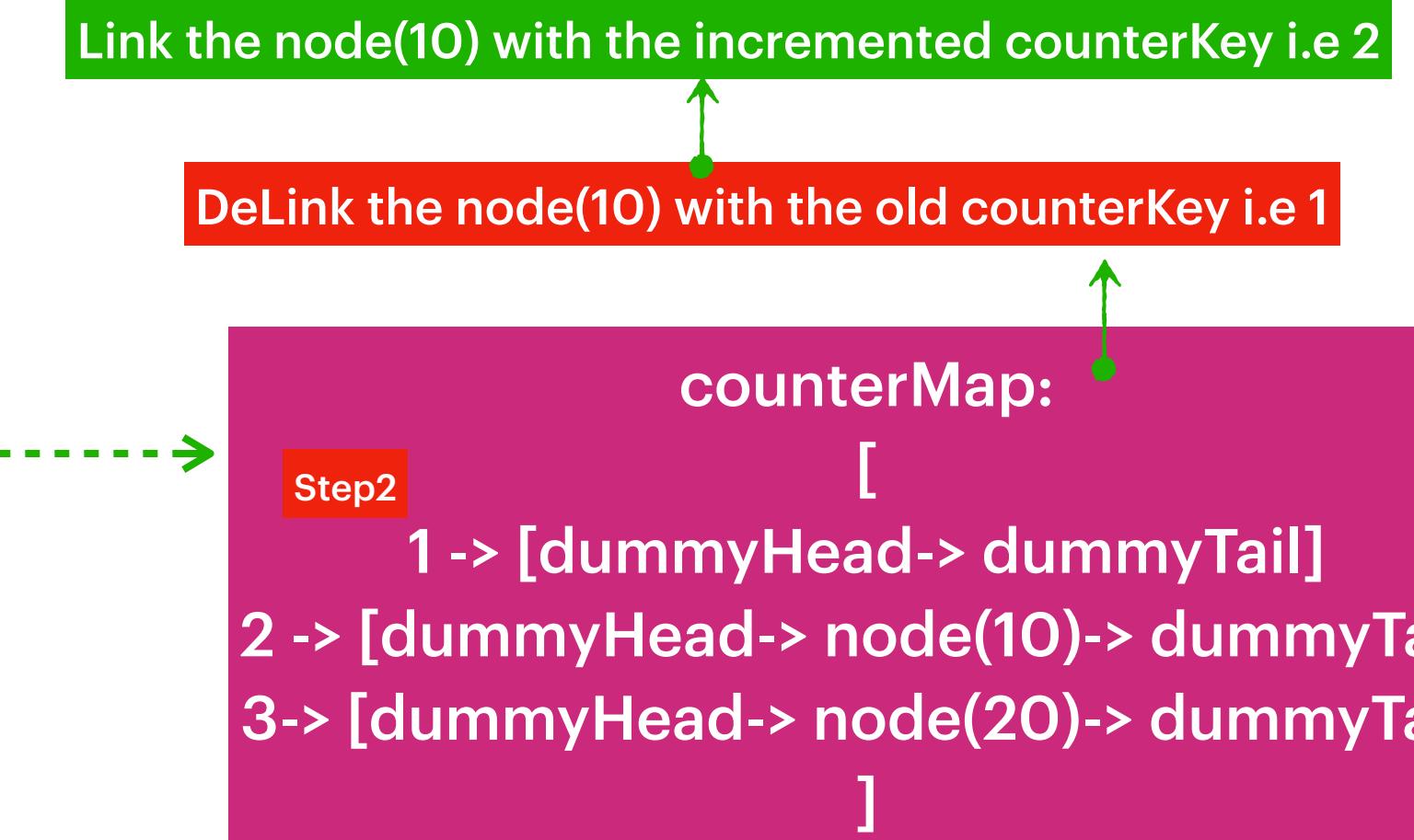
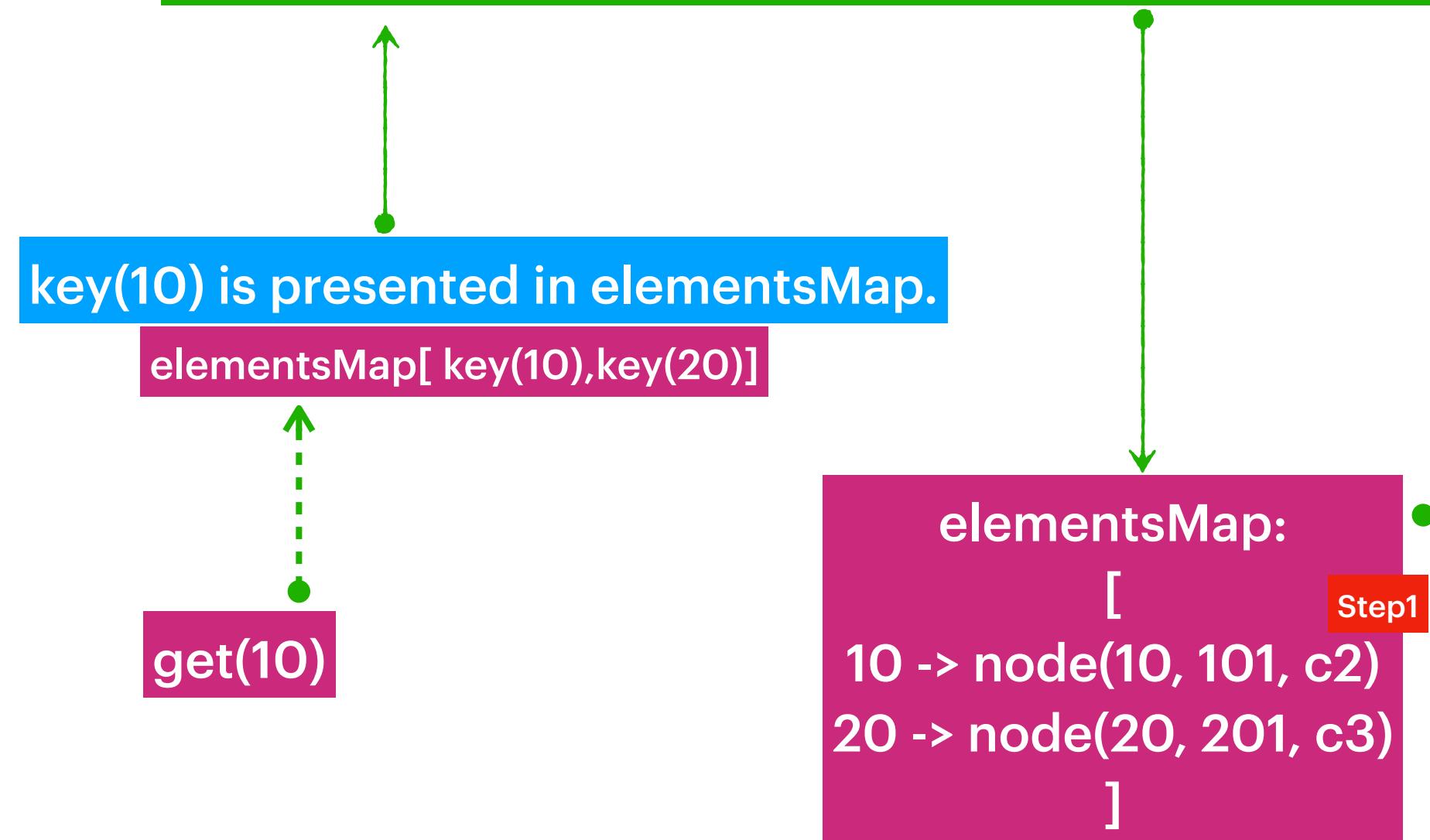


Step1: increment the counter , replace value of node(20).  
 Step2: In counterMap, DeLink the node(20) with the old counterKey i.e 2  
 Step3: Link the node(20) with the incremented counterKey i.e 3

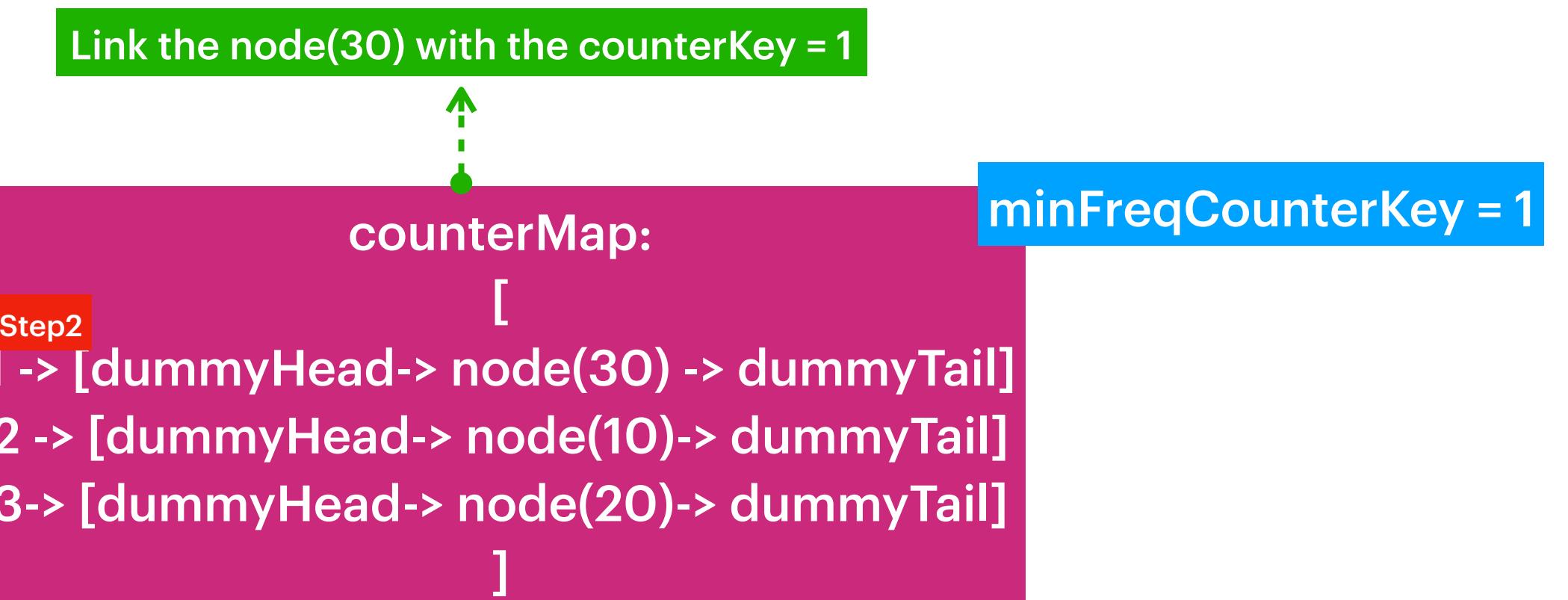
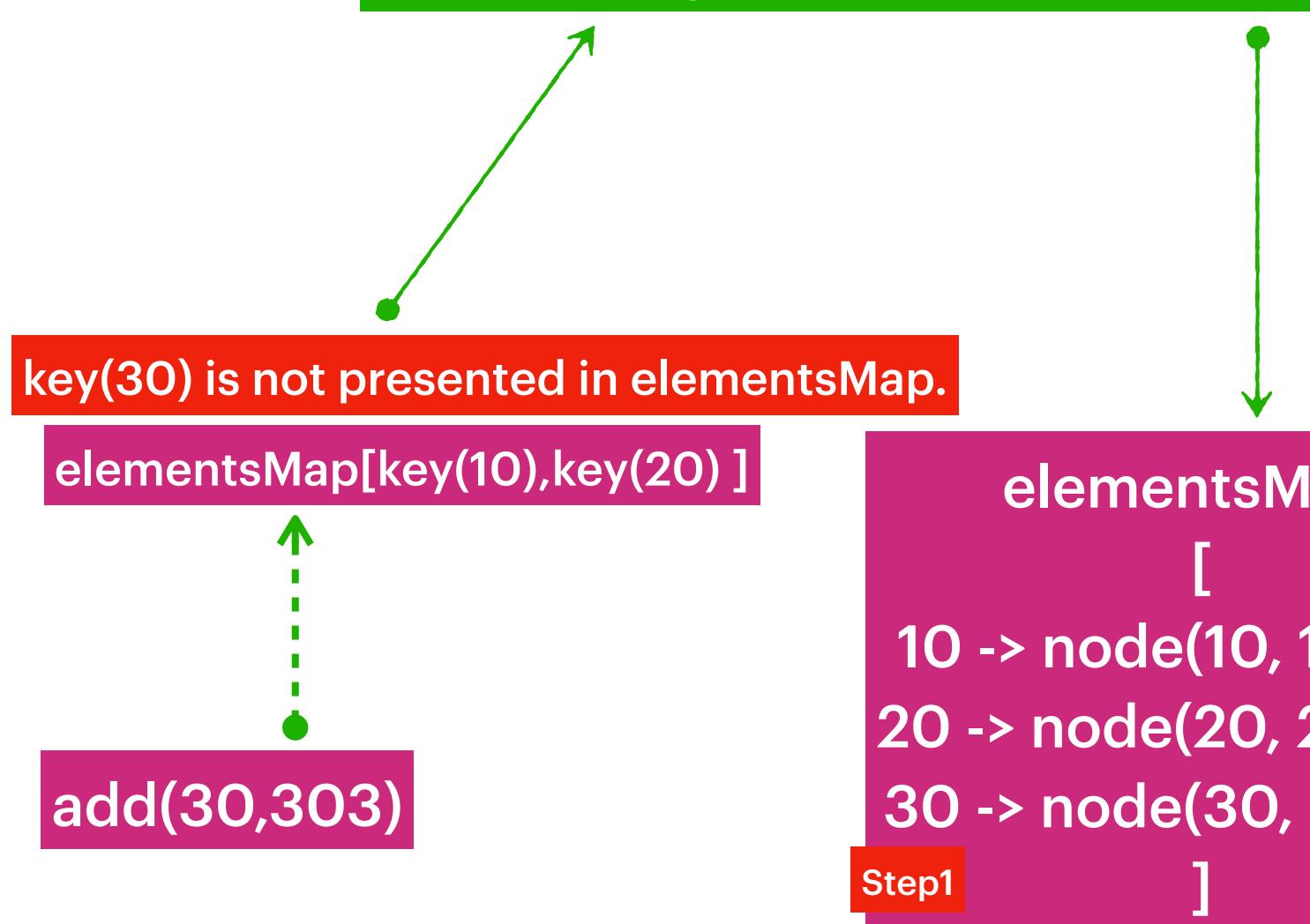


## LFUCache(capacity: 3)

Step1: increment the counter for node(10).  
 Step2: In counterMap, DeLink the node(10) with the old counterKey i.e 1  
 Step3: Link the node(10) with the incremented counterKey i.e 2



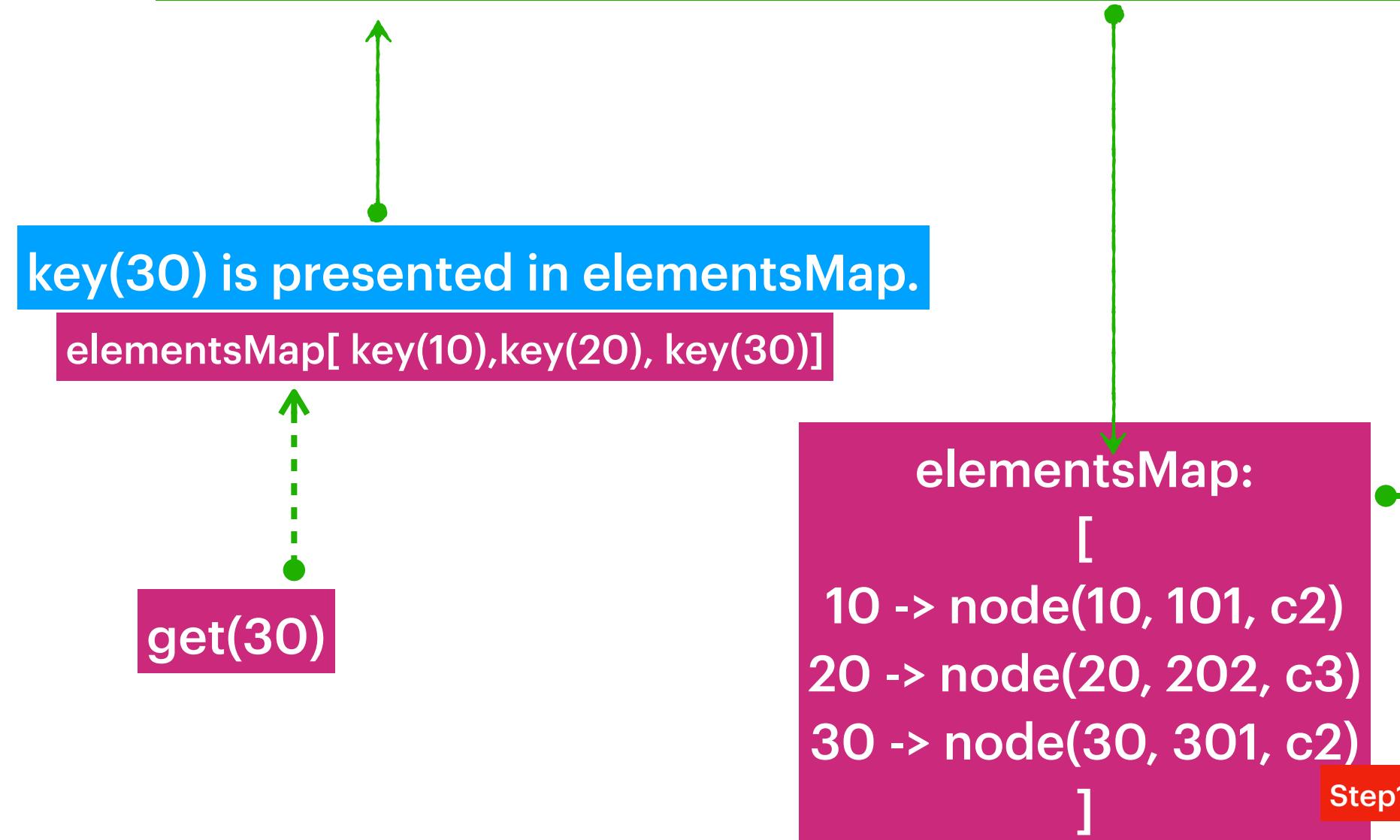
Step1: Add node(30) to the elementMap with key : 30 & initial counterValue =1,  
 Step2: Link node(30) in counterMap for the counterKey=1



**minFreqCounterKey should be 2:**  
 As we  
 don't have elements  
 with the counterKey:1

## LFUCache(capacity: 3)

Step1: increment the counter for node(30).  
 Step2: In counterMap, DeLink the node(30) with the old counterKey i.e 1  
 Step3: Link the node(30) with the new counterKey i.e 2



Link the node(10) with the incremented counterKey i.e 2

DeLink the node(10) with the old counterKey. i.e counter=1

`minFreqCounterKey = 2`

`minFreqCounterKey` should be 2:  
As we don't have elements with the counterKey:1

Adding new node to the right of dummyHead is the Concept of LRU Cache so that we can consider `dummyTail.prev` as old LFU element.

## LFUCache(capacity: 3)

### Removing of LFU element

If we add node(40) then size(4) > capacity (3)  
So First Remove LFU element :  
Then add node(40)

If closely see the earlier screen we have minFreqCounterKey: 2  
Which has 2 LFU nodes [dummyHead-> node(30)-> node(10)-> dummyTail]  
We would need to remove old LFU element i.e node(10).  
Remove node(10) from both elementsMap & counterMap

key(40) is not presented in elementsMap.

elementsMap[ key(10),key(20), key(30)]

add(40,401)

elementsMap:  
[  
20 -> node(20, 202, c3)  
30 -> node(30, 301, c2)  
]

counterMap:  
[  
1 -> [dummyHead-> dummyTail]  
2 -> [dummyHead-> node(30)--> dummyTail]  
3-> [dummyHead-> node(20)-> dummyTail]  
]

minFreqCounterKey = 2

Now we can add node(40) , So after adding node(40)

elementsMap:  
[  
20 -> node(20, 202, c3)  
30 -> node(30, 301, c2)  
40 -> node(40, 401, c1)  
]

counterMap:  
[  
1 -> [dummyHead—> node(40)—>dummyTail]  
2 -> [dummyHead-> node(30)--> dummyTail]  
3-> [dummyHead-> node(20)-> dummyTail]  
]

minFreqCounterKey = 1

## LFUCache(capacity: 3)

### Removing of LFU element

If we add node(50) then size(4) > capacity (3)  
So First Remove LFU element :  
Then add node(50)

If we closely observe the earlier screen, then we have minFreqCounterKey:1  
Which has 1 LFU node [dummyHead-> node(40)-> dummyTail]  
We would need to remove LFU element, node(40).  
Remove node(40) from both elementsMap & counterMap

key(50) is not presented in elementsMap.

elementsMap[ key(10),key(20), key(30)]

add(50,501)

elementsMap:  
[  
20 -> node(20, 202, c3)  
30 -> node(30, 301, c2)  
]

counterMap:  
[  
1 -> [dummyHead-> dummyTail]  
2 -> [dummyHead-> node(30)--> dummyTail]  
3-> [dummyHead-> node(20)-> dummyTail]  
]

minFreqCounterKey = 2

minFreqCounterKey  
should be 2:  
As we  
don't have elements  
with the counterKey:1

Now we can add node(50) , So after adding node(50)

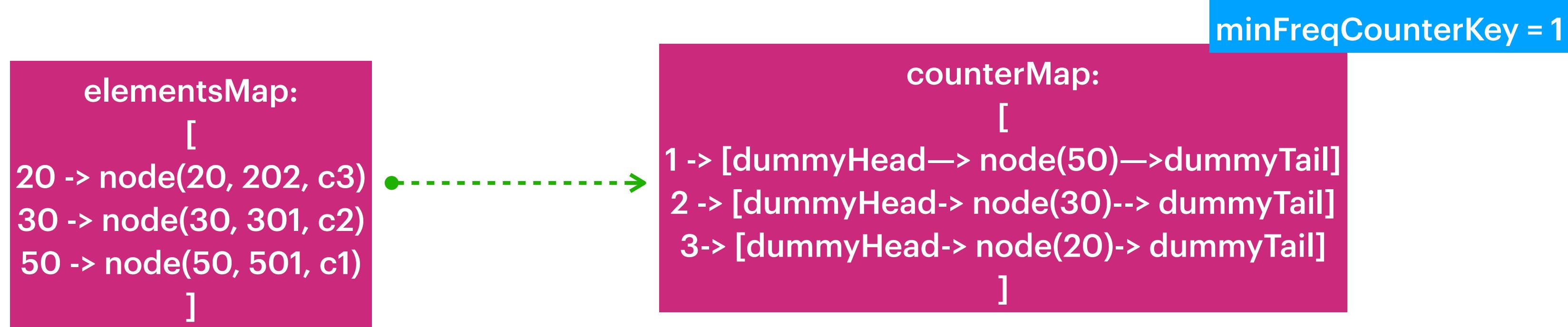
elementsMap:  
[  
20 -> node(20, 202, c3)  
30 -> node(30, 301, c2)  
50 -> node(50, 501, c1)  
]

counterMap:  
[  
1 -> [dummyHead-> node(50)--> dummyTail]  
2 -> [dummyHead-> node(30)--> dummyTail]  
3-> [dummyHead-> node(20)-> dummyTail]  
]

minFreqCounterKey = 1

minFreqCounterKey  
Updated to 1:  
As the latest  
elementCount (1)  
<  
earlierOne(2).

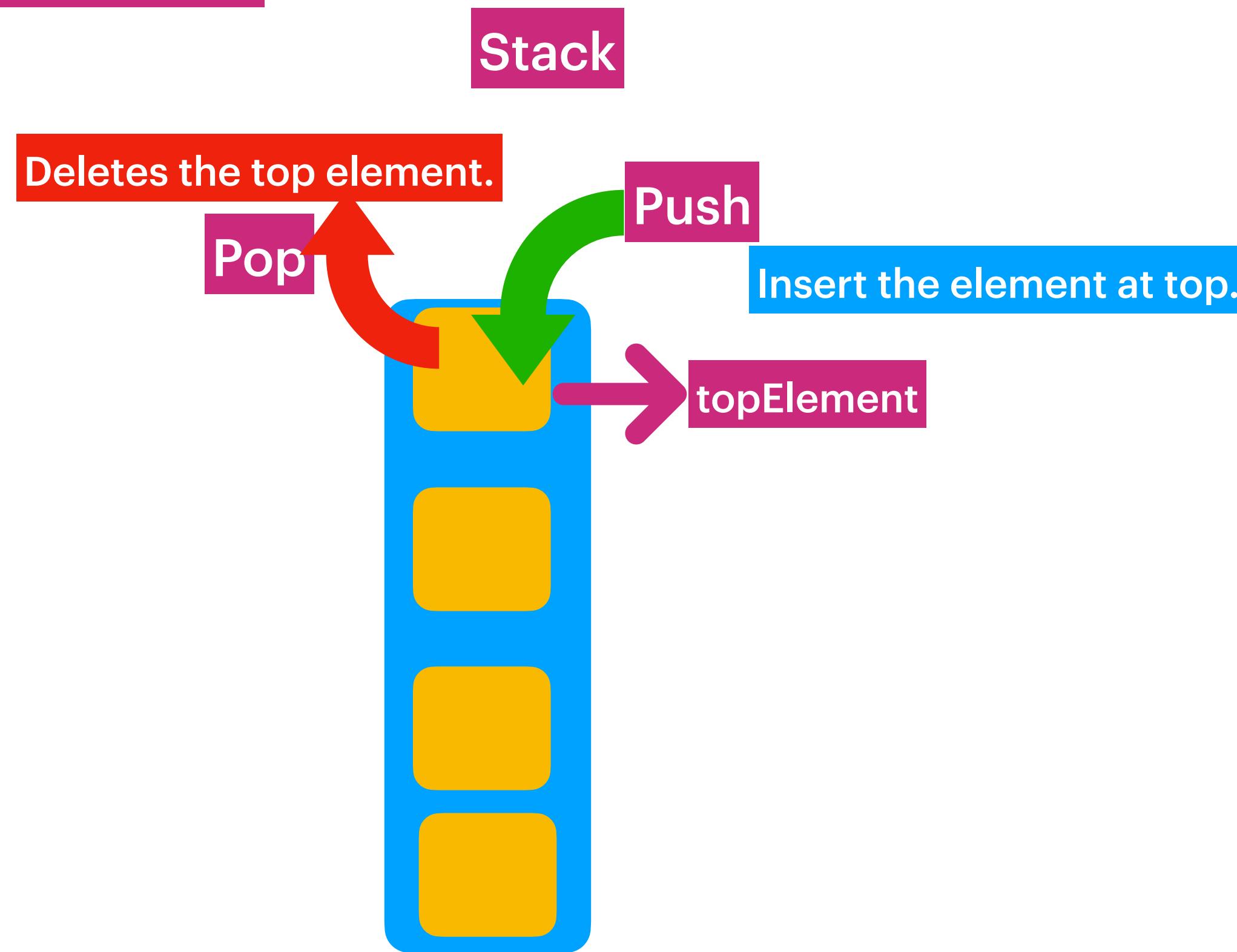
## LFUCache(capacity: 3)



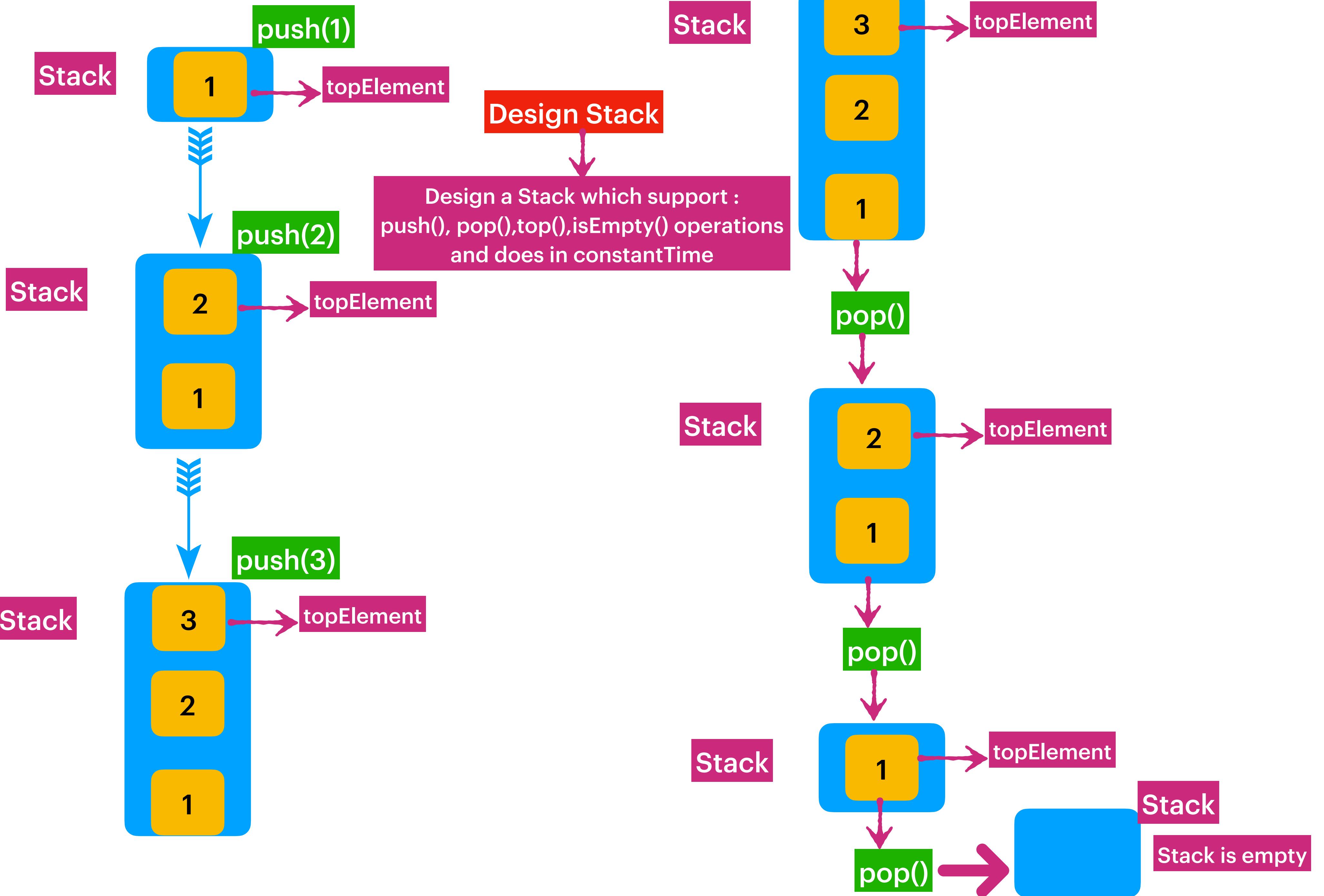
Finally Output of following get Calls



## Know about Stack

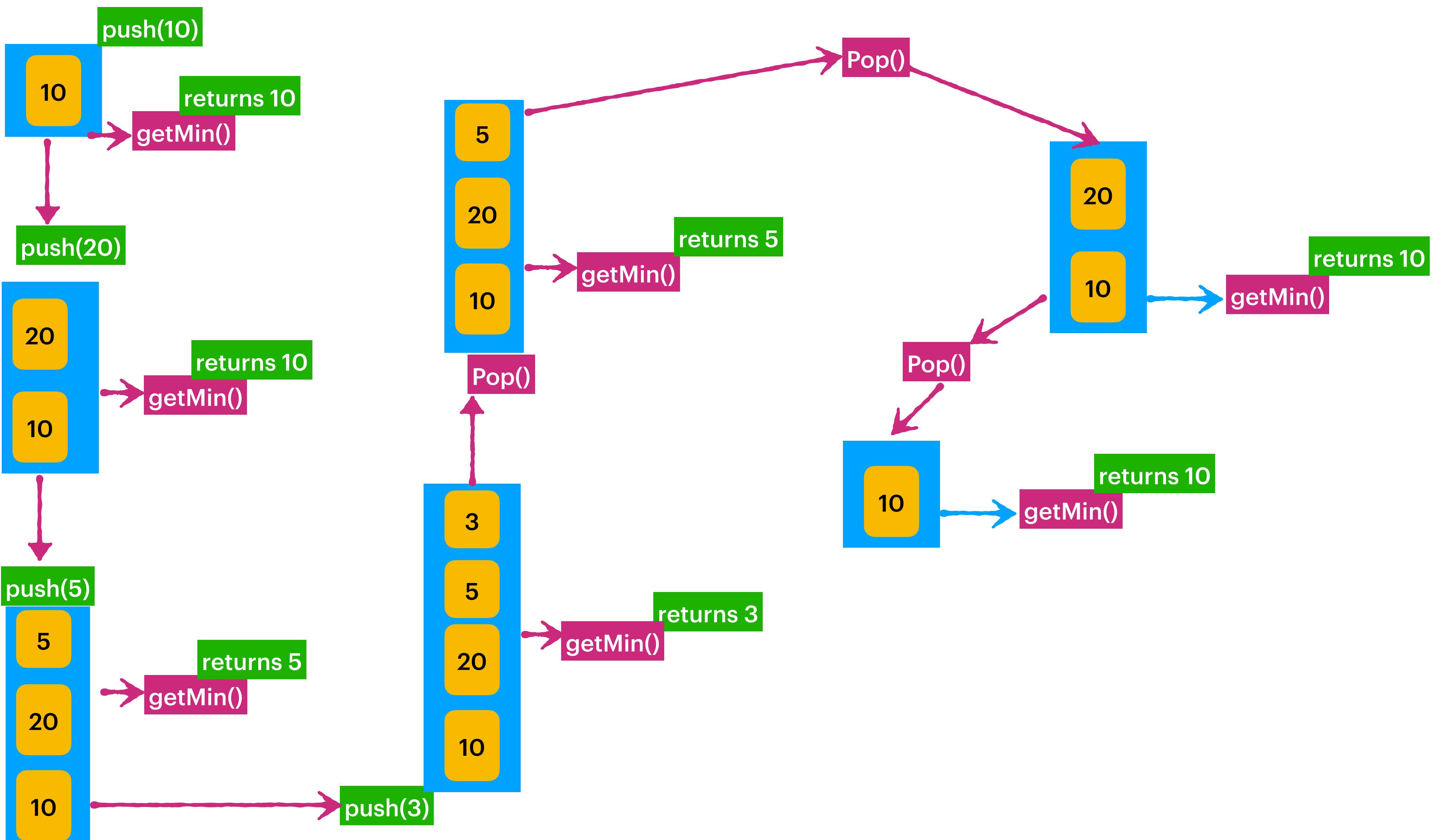


=> Stack follows Last In First Out order. (LIFO)  
LIFO => Last Inserted Element can be accessed/deleted first  
=> Stack always inserts the element at end.  
=> Stack always deletes the top element.



## Design Min Stack

Design a stack that supports push, pop, peek, and retrieving the minimum element in constant time.



## Valid Parentheses

Given a string **s** containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

- 1) Open brackets must be closed by the same type of brackets.
- 2) Open brackets must be closed in the correct order.

**Input:** s = "()"  
**Output:** true

**Input:** s = "()"  
**Output:** false

**Input:** s = "()[]{}"  
**Output:** true

**Input:** s = "([{}])"  
**Output:** false

**Input:** s = "[{}]"  
**Output:** true

**Input:** s = "())"  
**Output:** false

### Daily Temperatures :

Given an input array with each day temperature, find the next warm day from current day.  
If you don't find next warm day then put as zero.

**Input:**

25    5    30    25    40

**Output:**

2    1    2    1    0

**Input:**

29    20    19    21    30    29    30

**Output:**

4    2    1    1    0    1    0

**Input:**

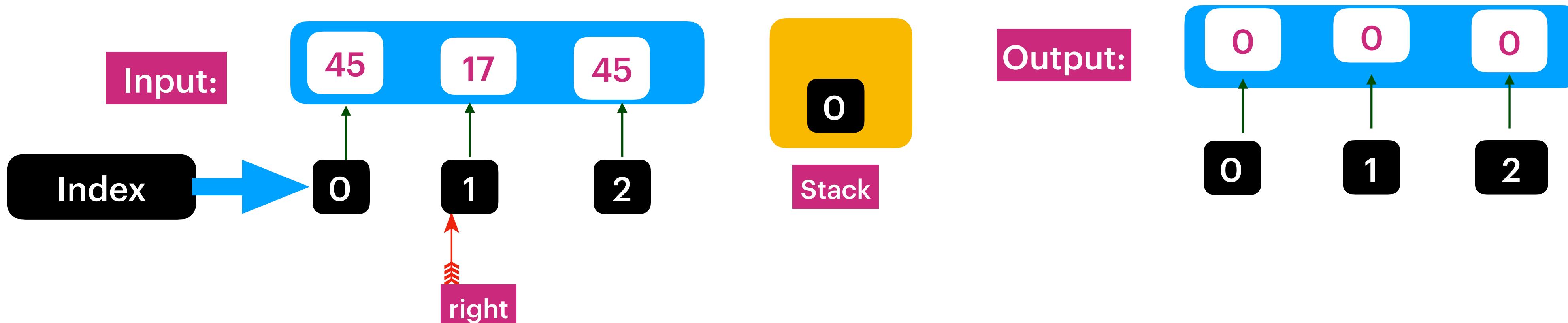
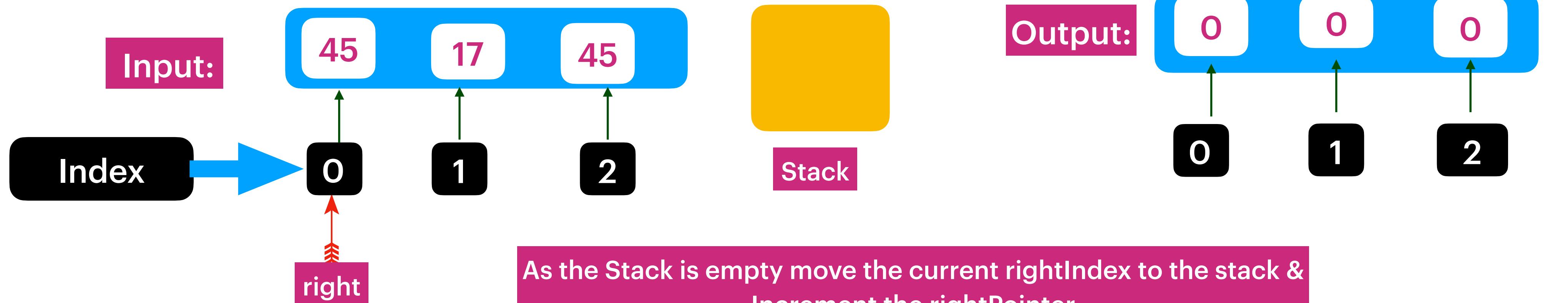
70    60    50    40    30

**Output:**

0    0    0    0    0

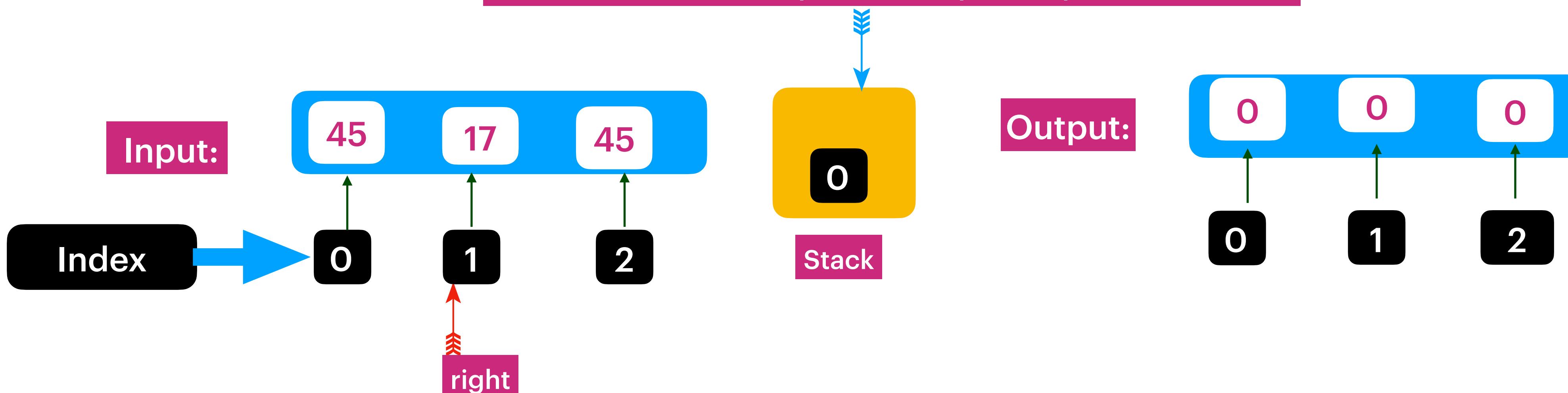
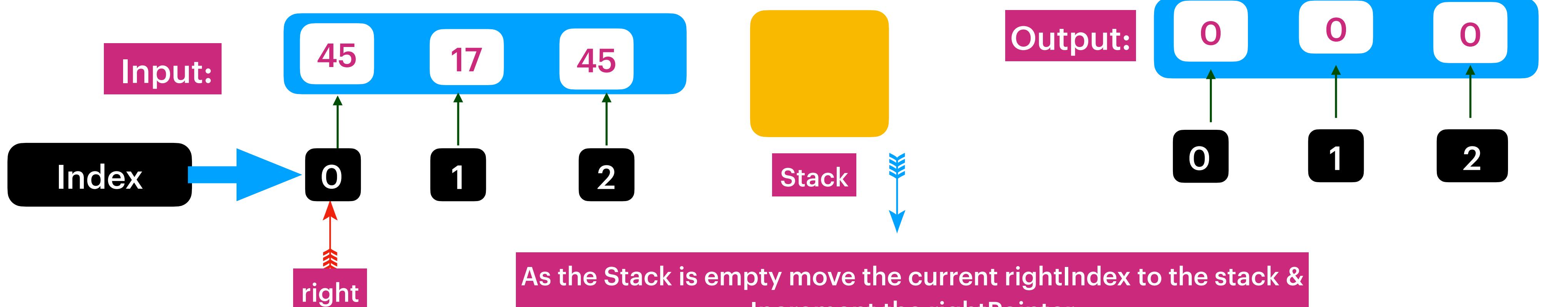
## Increasing / Decreasing Stack : (Monotonic Stack Pattern)

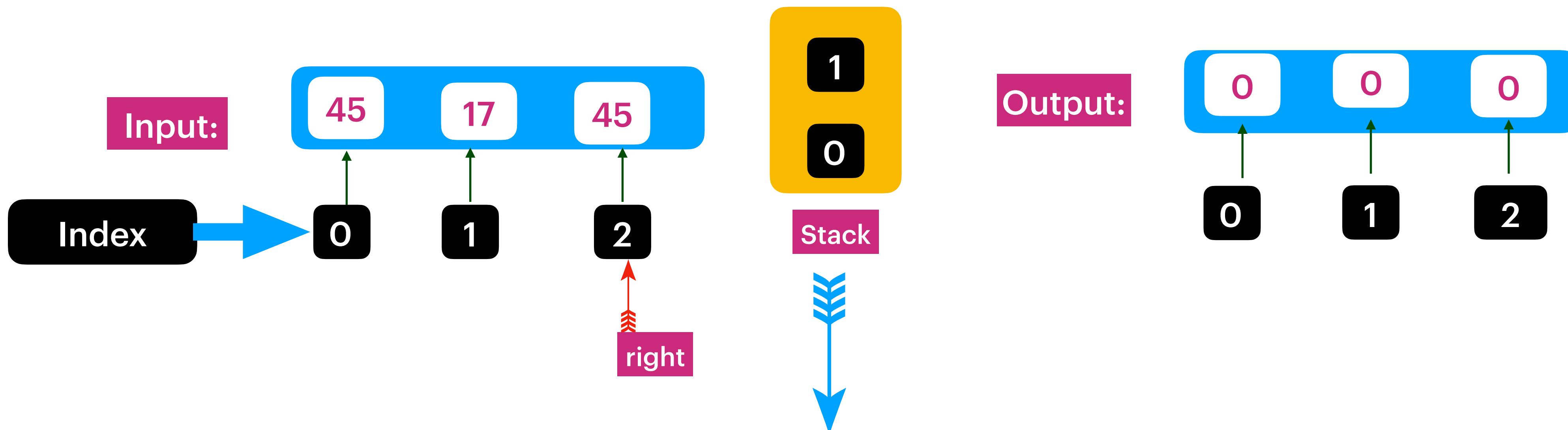
Initiate the output array of same input size with default value as 0



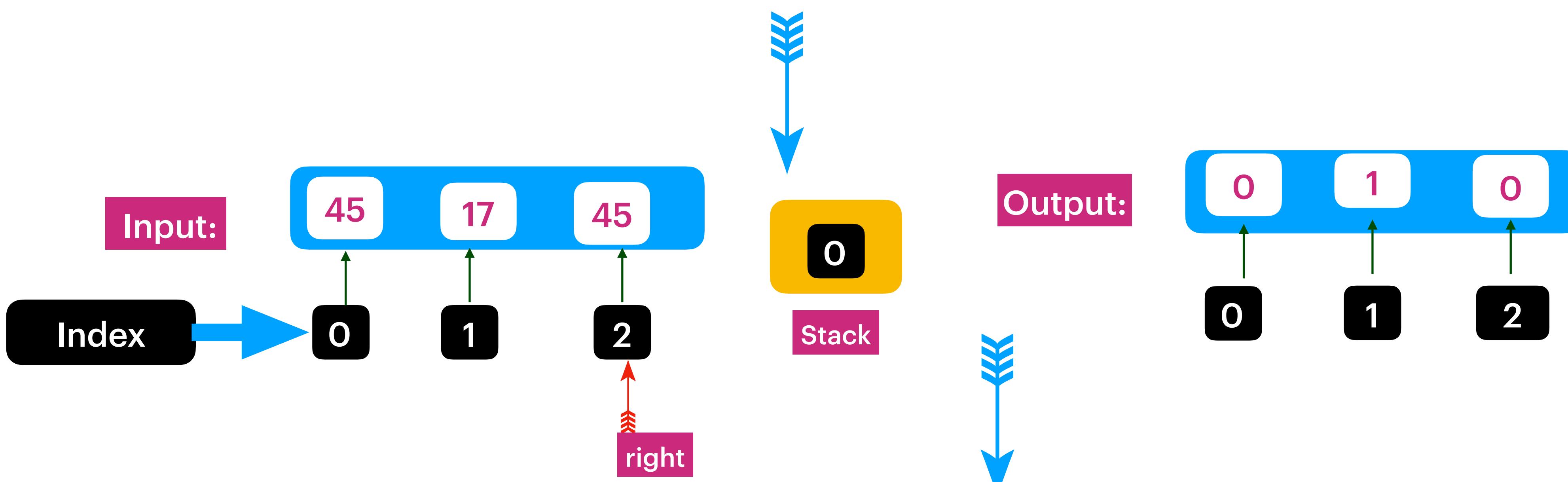
## Solution with Increasing / Decreasing Stack : (Monotonic Stack Pattern)

Initiate the output array of same input size with default value as 0

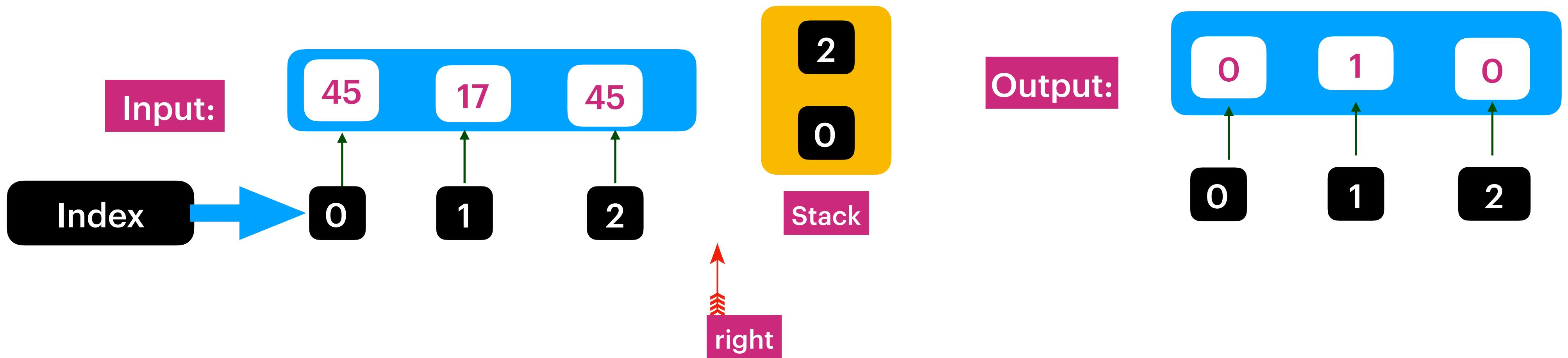




As my currentRight (45) > top element(17) of the stack. i.e input[1] = 15  
 Pop the topIndex from the stack i.e 1  
 Update the output array :  
 $\text{output}[\text{popIndex}] = \text{currentRightIndex} - \text{PopIndex}$   
 $\text{output}[1] = 2 - 1$

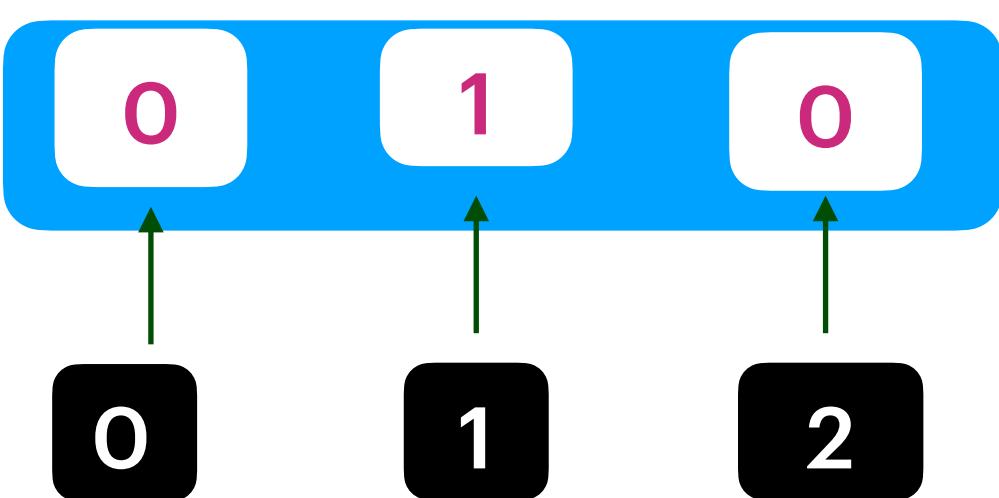


As my currentRight (45) is not greater than top element(45) of the stack.  
Push the current index to stack & increment the rightPointer.



Now rightPointer reaching out of array size so we can return OutPut Array i.e

In worst cast each character would be visited twice :  
1) By rightPointer in forward direction  
2) Through stack in backward Direction  
So the TimeComplexity would be  $O(2n) = O(n)$   
Space Complexity :  $O(n)$



**Solution with Back Tracking, Its Math Solution, refer to the notes.**

## Next Greater Element :

Replace current Element in given input array with next greater element

Input:



Output:



Input:



Output:



## Design Online Stock Span:

Design an algorithm that collects daily price quotes for some stock and returns the span of that stock's price for the current day.

The span of the stock's price today is defined as the maximum number of consecutive days (starting from today and going backward) for which the stock price was less than or equal to today's price.

For example, if the price of a stock over the next 7 days were [100,80,60,70,60,75,85], then the stock spans would be [1,1,1,2,1,4,6].

Implement the StockSpanner class:

StockSpanner() Initializes the object of the class.

int next(int price) Returns the span of the stock's price given that today's price is price.

### Input

```
["StockSpanner", "next", "next", "next", "next", "next", "next", "next", "next"]
[], [100], [80], [60], [70], [60], [75], [85]
```

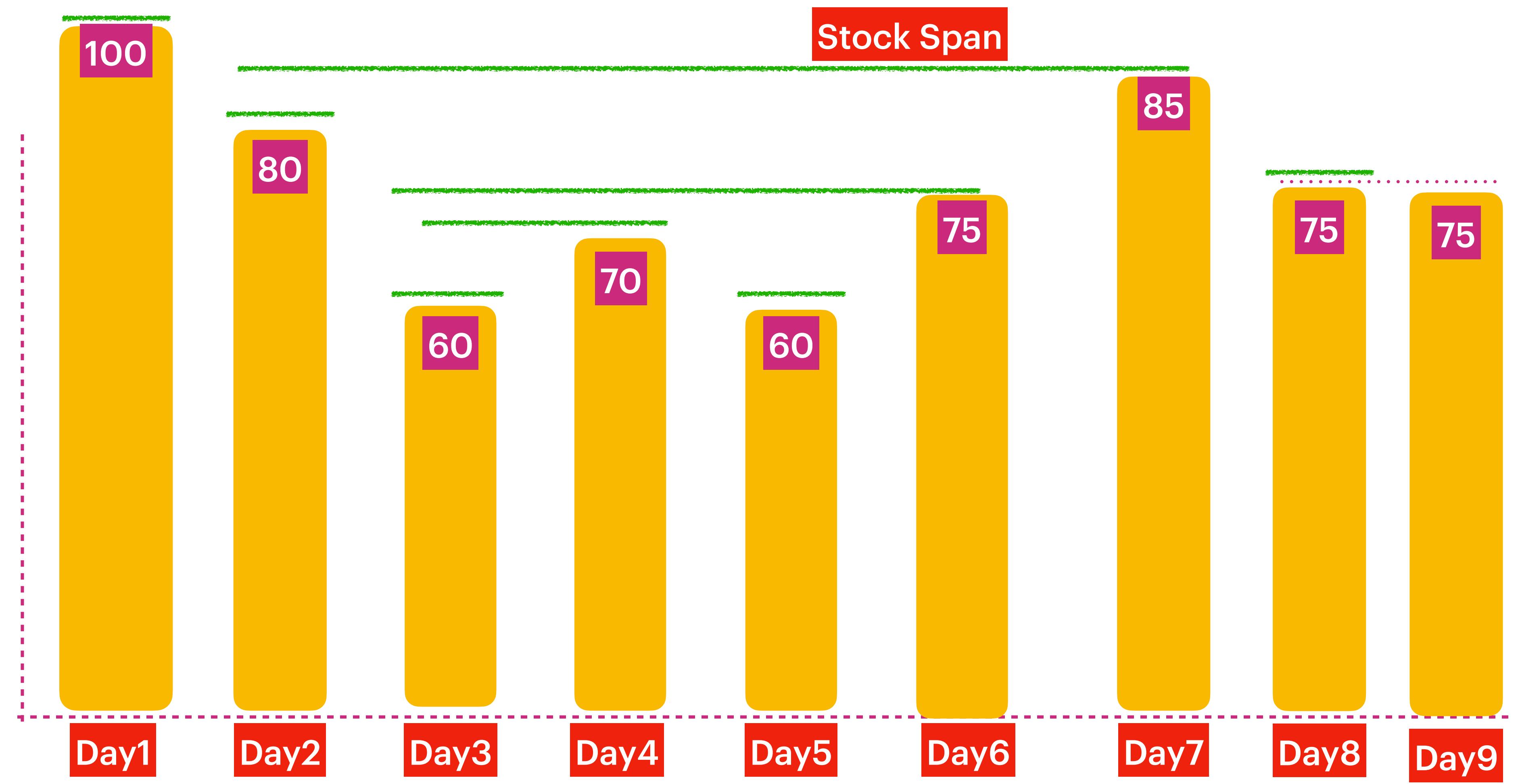
### Output

```
[null, 1, 1, 1, 2, 1, 4, 6]
```

### Explanation

```
StockSpanner stockSpanner = new StockSpanner();
stockSpanner.next(100); // return 1
stockSpanner.next(80); // return 1
stockSpanner.next(60); // return 1
stockSpanner.next(70); // return 2
stockSpanner.next(60); // return 1
stockSpanner.next(75); // return 4, because the last 4 prices (including today's price of 75) were less than or equal to today's price.
stockSpanner.next(85); // return 6
```

## Stock Span



Next 7 days were [100], [80], [60], [70],[60],[75],[85], [75] , [75]

Expected Output : [1] , [1], [1] , [2] , [1] ,[4] , [6], [1] ,[2]



Time  
Complexity

Completed As of Today



Space  
Complexity



Sorting &  
Searching



Dynamic  
Programming



Arrays &  
Recursion



Divide &  
Conquer



Starting From Monday  
18th Oct 2021

List

Hashing

Stack, Queues

Graphs

Math  
Patterns

Solving System  
Design problems with DS

Greedy  
Algorithms

Trees,  
Trie (Advanced Trees)

Heaps

## Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, \*, and /. Each operand may be an integer or another expression.

Note that division between two integers should truncate toward zero.

It is guaranteed that the given RPN expression is always valid. That means the expression would always evaluate to a result, and there will not be any division by zero operation.

**Input:** tokens = ["2","1","+", "3", "\*"]

**Output:** 9

**Explanation:**  $((2 + 1) * 3) = 9$

**Input:** tokens = ["10", "6", "9", "3", "+", "-11", "\*", "/", "\*", "17", "+", "5", "+"]

**Output:** 22

**Explanation:**  $((10 * (6 / ((9 + 3) * -11))) + 17) + 5$

$$= ((10 * (6 / (12 * -11))) + 17) + 5$$

$$= ((10 * (6 / -132)) + 17) + 5$$

$$= ((10 * 0) + 17) + 5$$

$$= (0 + 17) + 5$$

$$= 17 + 5$$

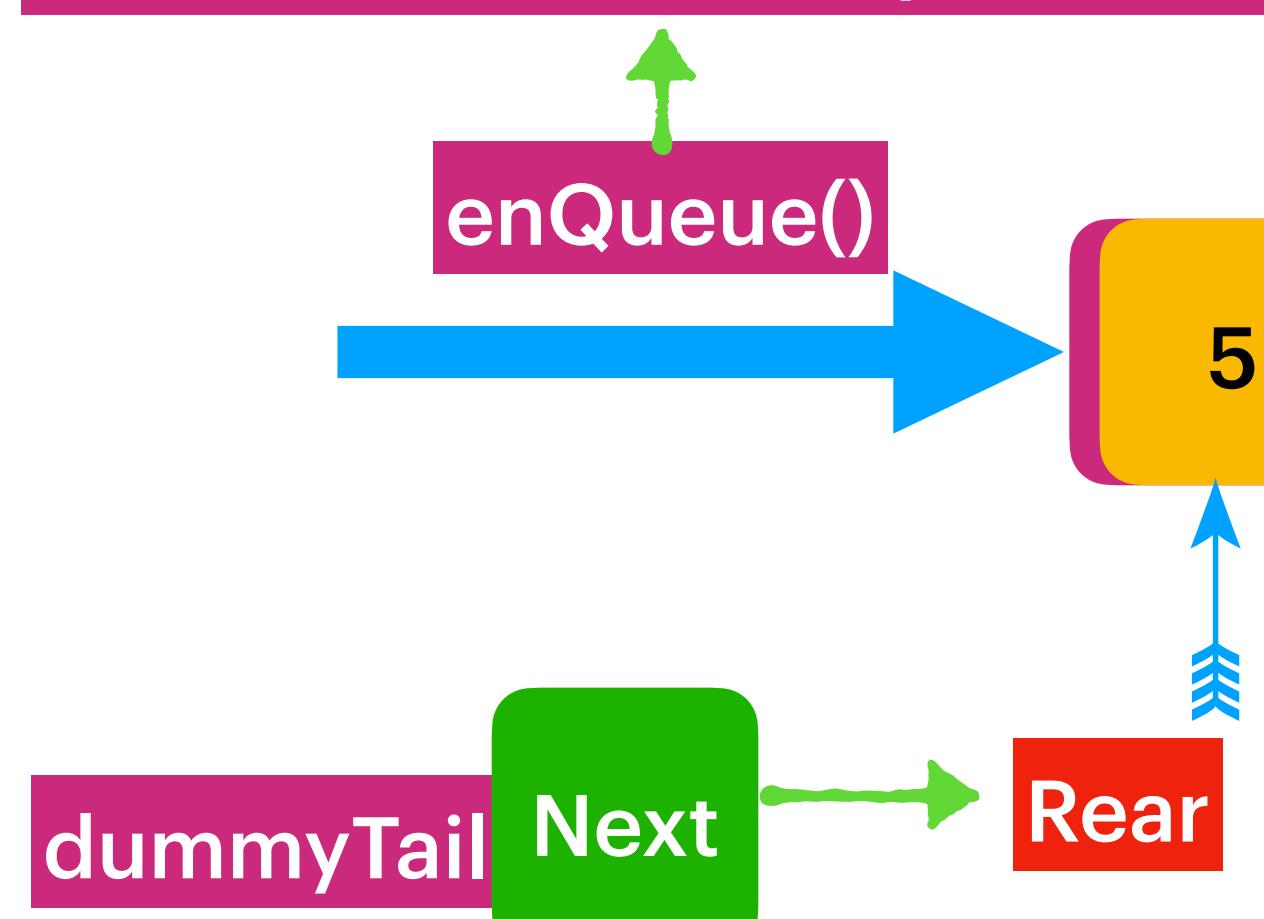
$$= 22$$

**Input:** tokens = ["4", "13", "5", "/", "+"]

**Output:** 6

**Explanation:**  $(4 + (13 / 5)) = 6$

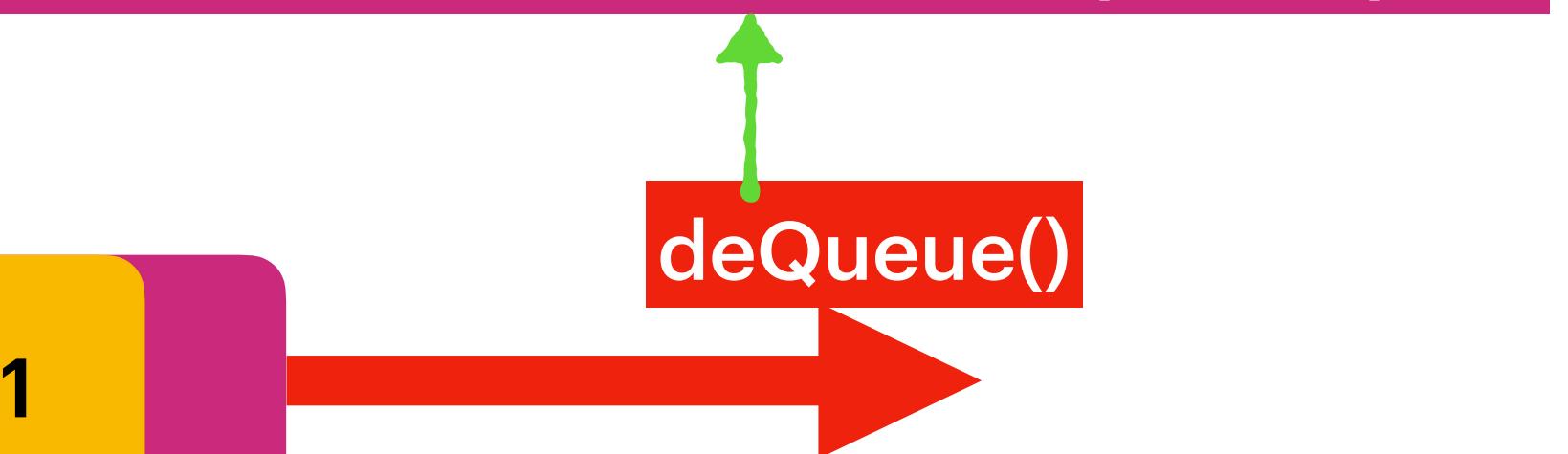
Adds to Rear : dummyTail.next



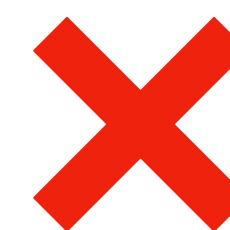
Design Queue

First In First Out

Removes from front : dummyHead.prev



ArrayList :



SingleLinkedList  
Can be possible put extra effort for prev reference

LinkedList

DoubleLinkedList ✓

Queue

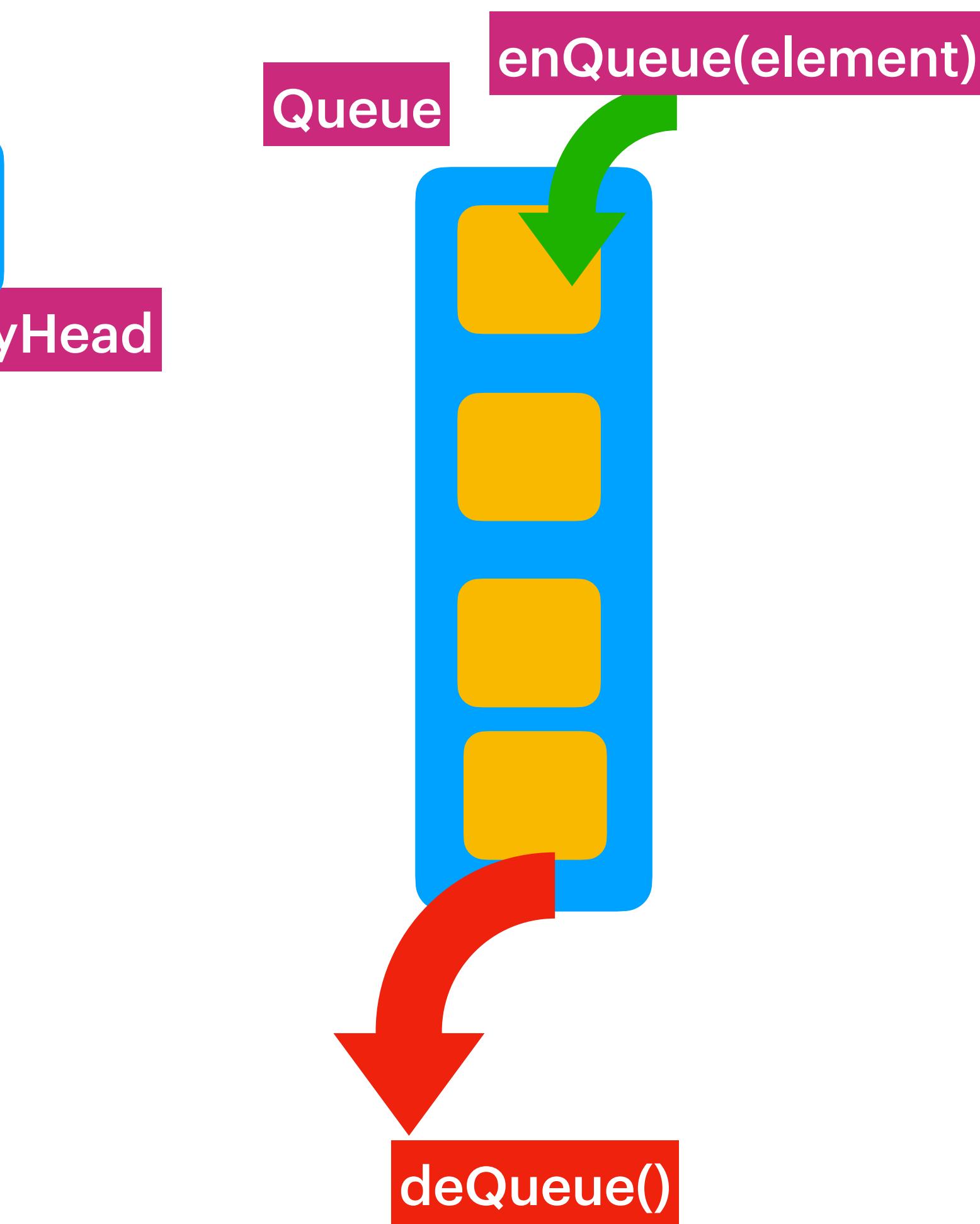
enQueue(element)

deQueue()

getFirst()

getLast()

isEmpty()



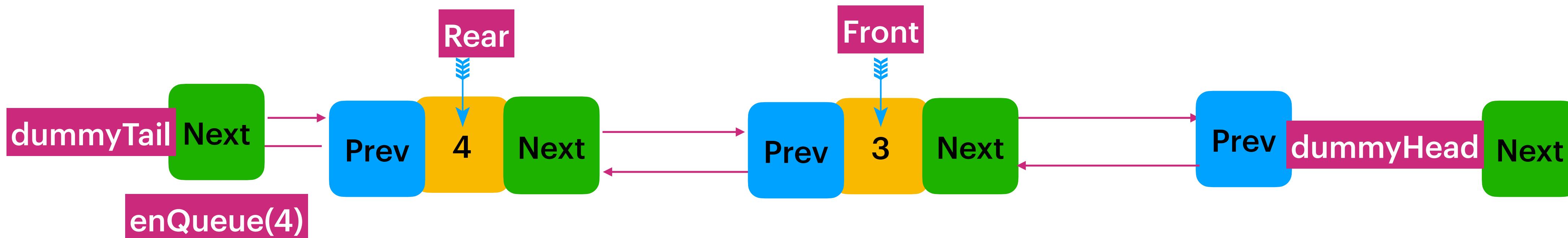
Represents Empty Queue



enQueue(3)



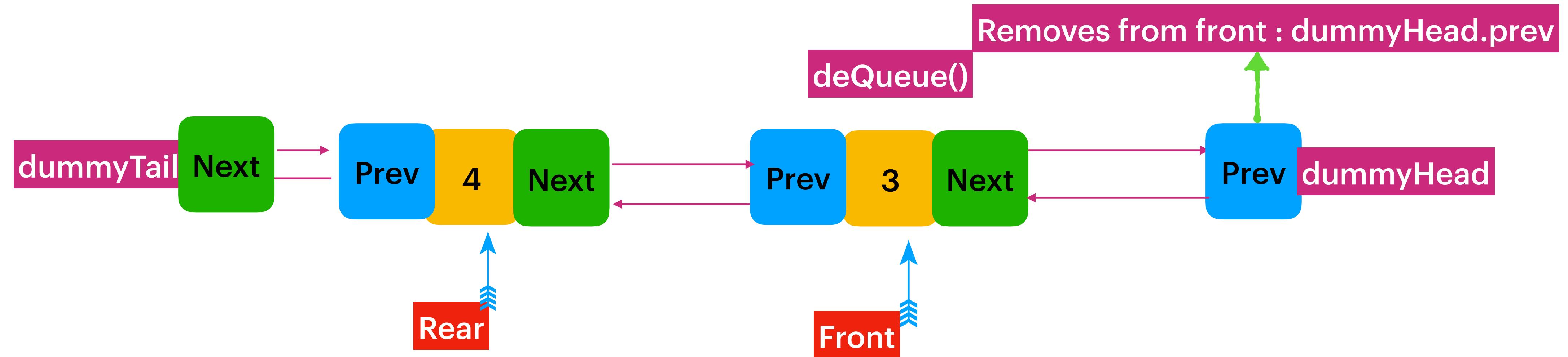
enQueue(4)



We add to the tail :

```
DLLNode tailNext = dummyTail.next;  
DLLNode current = new DLLNode(value);  
    current.next = tailNext;  
    tailNext.prev = current;  
    dummyTail.next = current;  
    current.prev = dummyTail;
```

enQueue(element) —>  
Adds to **dummyTail** i.e **dummyTail.next**



**deQueue() —> Removes from front : i.e dummyHead.prev**

```
DLLNode headPrev = dummyHead.prev;
dummyHead.prev = headPrev.prev
headPrev.prev.next = dummyHead

headPrev.next = null;
headPrev.prev = null; // Helps GC
```

Queue

DeQueue

Graph

Dec

Trees/Heaps

45 Days

Dynamic Programming / Greedy Algos / BackTracking

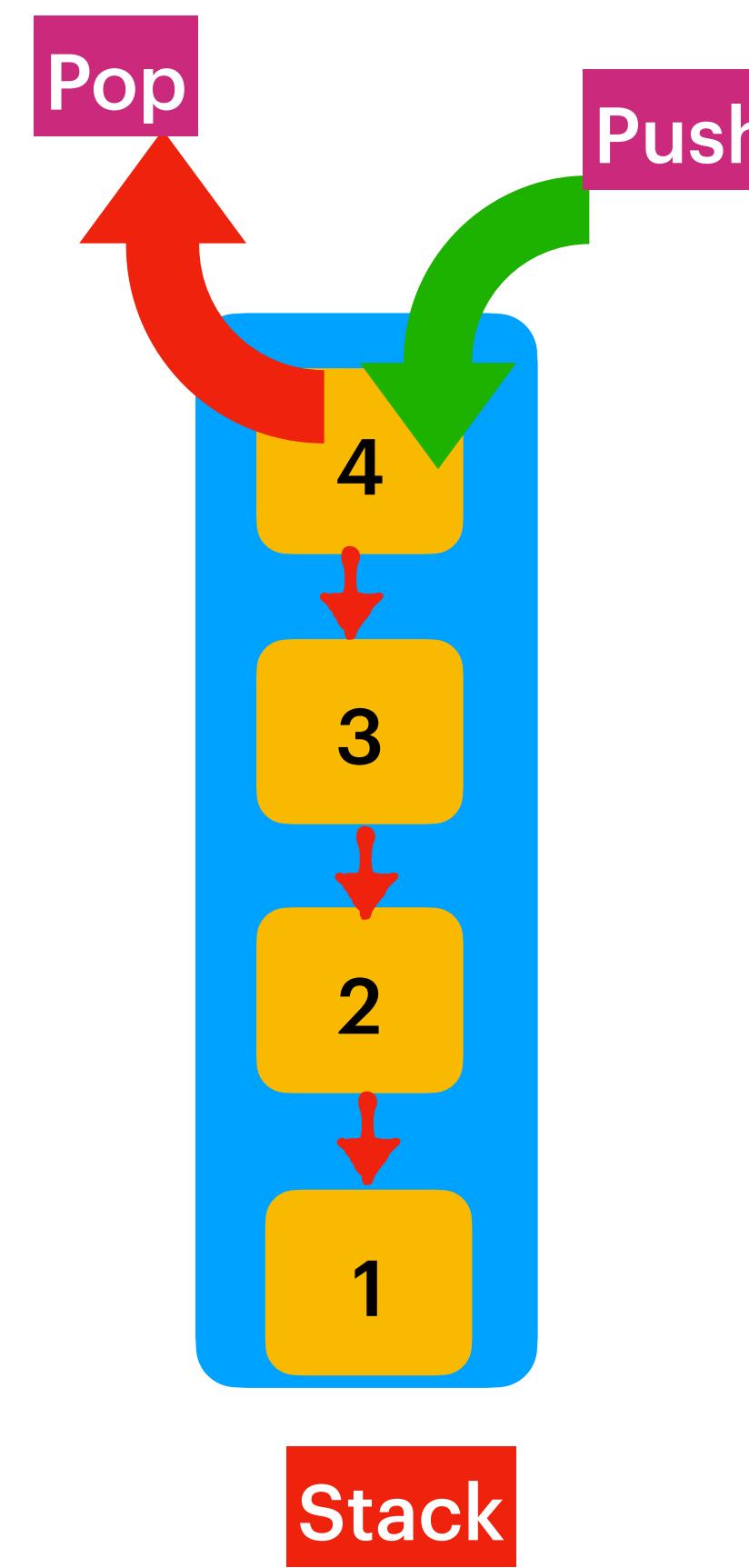
15 Problems

45 Days

20 Days Leet Code

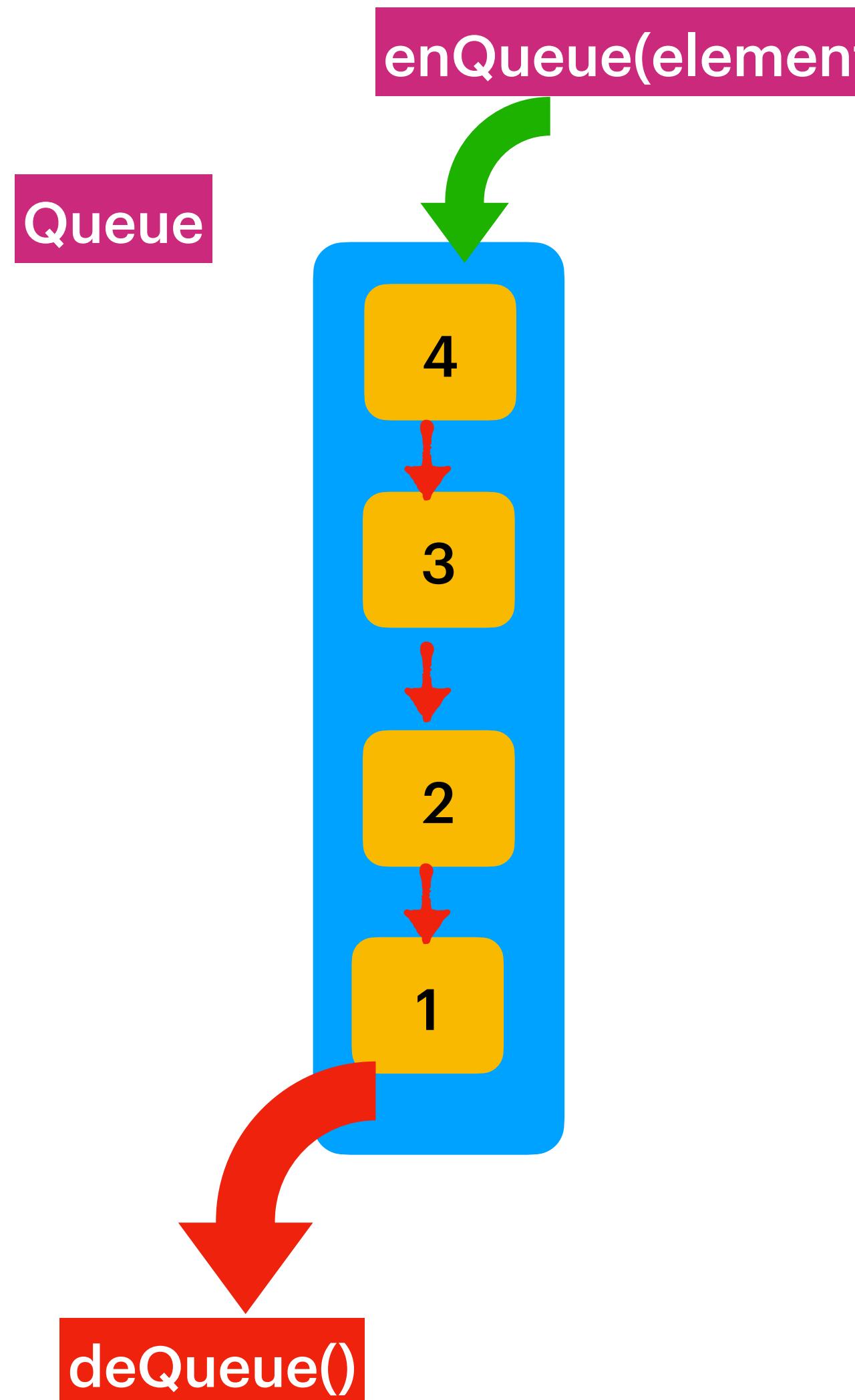
15 Problems

## Design Queue Using Stack

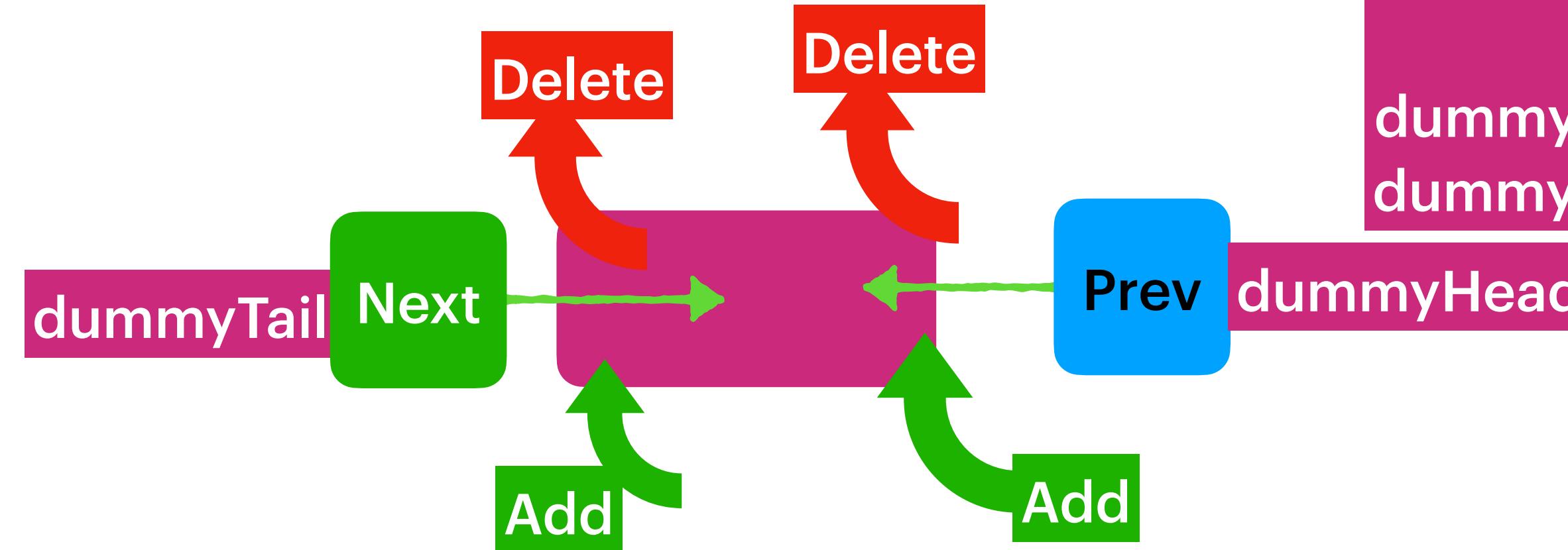


`top()`  
`enQueue()`  
`deQueue()`  
`isEmpty()`

## Design Stack Using Queue

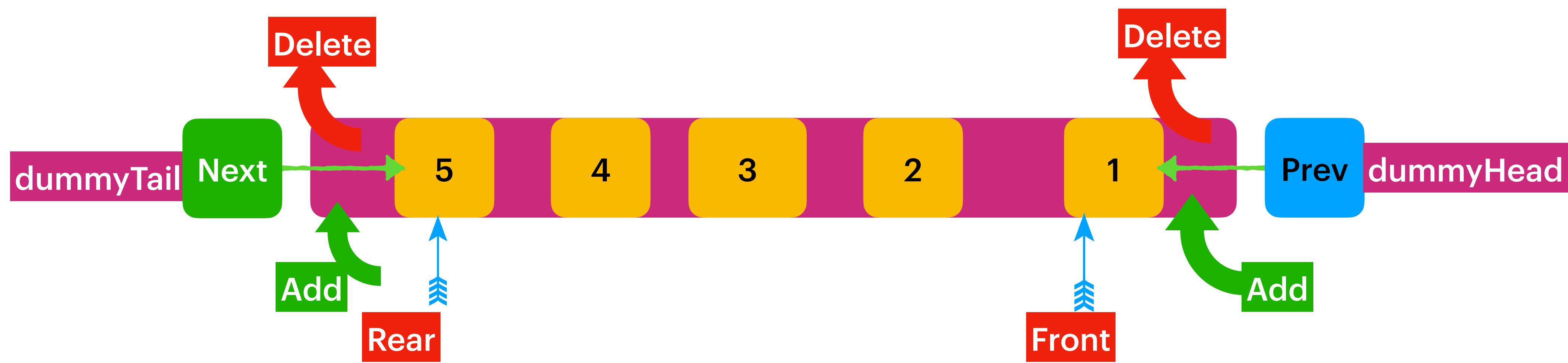


**Deque: Its a double ended queue.**  
**We can insert and delete from front & rear.**



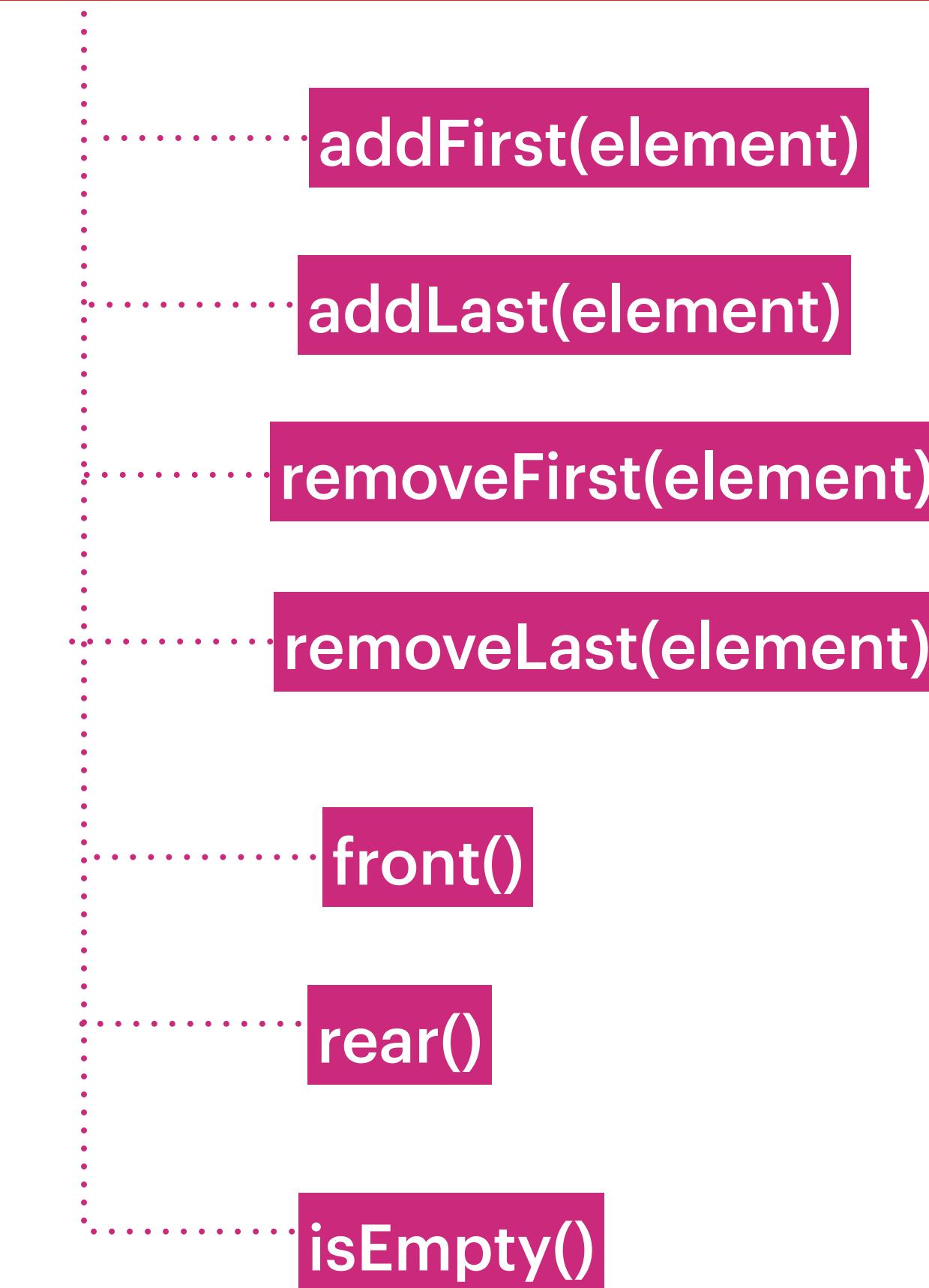
**It Represents Empty Deque:**

```
dummyTail.next = dummyHead;  
dummyHead.prev = dummyTail;
```



# Design Deque :

## It's a double ended Queue. We can add, remove from front & rear.



## Moving Average from Data Stream

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

Implement the MovingAverage class:

**MovingAverage(int size)** Initializes the object with the size of the window size.  
**double next(int val)** Returns the moving average of the last size values of the stream.

Input

```
["MovingAverage", "next", "next", "next", "next"]
[[3], [1], [10], [3], [5]]
```

Output

```
[null, 1.0, 5.5, 4.66667, 6.0]
```

Explanation

```
MovingAverage movingAverage = new MovingAverage(3);
movingAverage.next(1); // return 1.0 = 1 / 1
movingAverage.next(10); // return 5.5 = (1 + 10) / 2
movingAverage.next(3); // return 4.66667 = (1 + 10 + 3) / 3
movingAverage.next(5); // return 6.0 = (10 + 3 + 5) / 3
```

## Jump Game VI

You are given a 0-indexed integer array `nums` and an integer `k`.

You are initially standing at index 0. In one move, you can jump at most `k` steps forward without going outside the boundaries of the array.  
That is, you can jump from index `i` to any index in the range  $[i + 1, \min(n - 1, i + k)]$  inclusive.

You want to reach the last index of the array (index `n - 1`). Your score is the sum of all `nums[j]` for each index `j` you visited in the array.

Return the maximum score you can get.

**Input:** `nums = [1,-1,-2,4,-7,3]`, `k = 2`

**Output:** 7

**Explanation:** You can choose your jumps forming the subsequence `[1,-1,4,3]` (underlined above). The sum is 7.

**Input:** `nums = [10,-5,-2,4,0,3]` `k = 3`

**Output:** 17

**Explanation:** You can choose your jumps forming the subsequence `[10,4,3]`. The sum is 17.

**Input:** nums = [10,-5,-2,4,0,3] k = 3

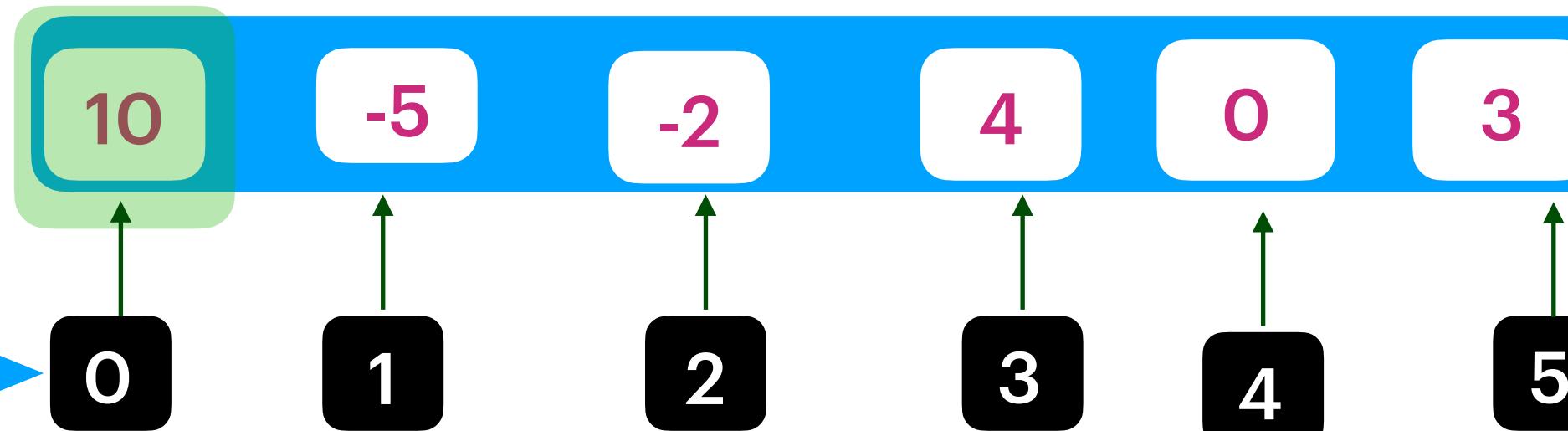
**Output:** 17

**Explanation:** You can choose your jumps forming the subsequence [10,4,3]. The sum is 17.

### BruteForce Approach

maxScore = 10

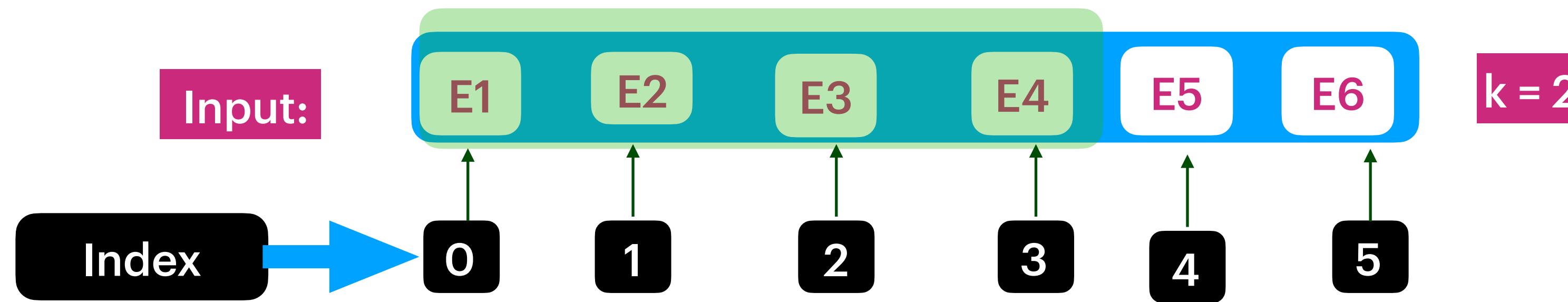
Input:



Index

k = 2

10  
10 - 5 = 5  
10 - 2 = 8  
8  
8 + 4 = 12  
8 + 0 = 8  
12  
12 + 3 = 15



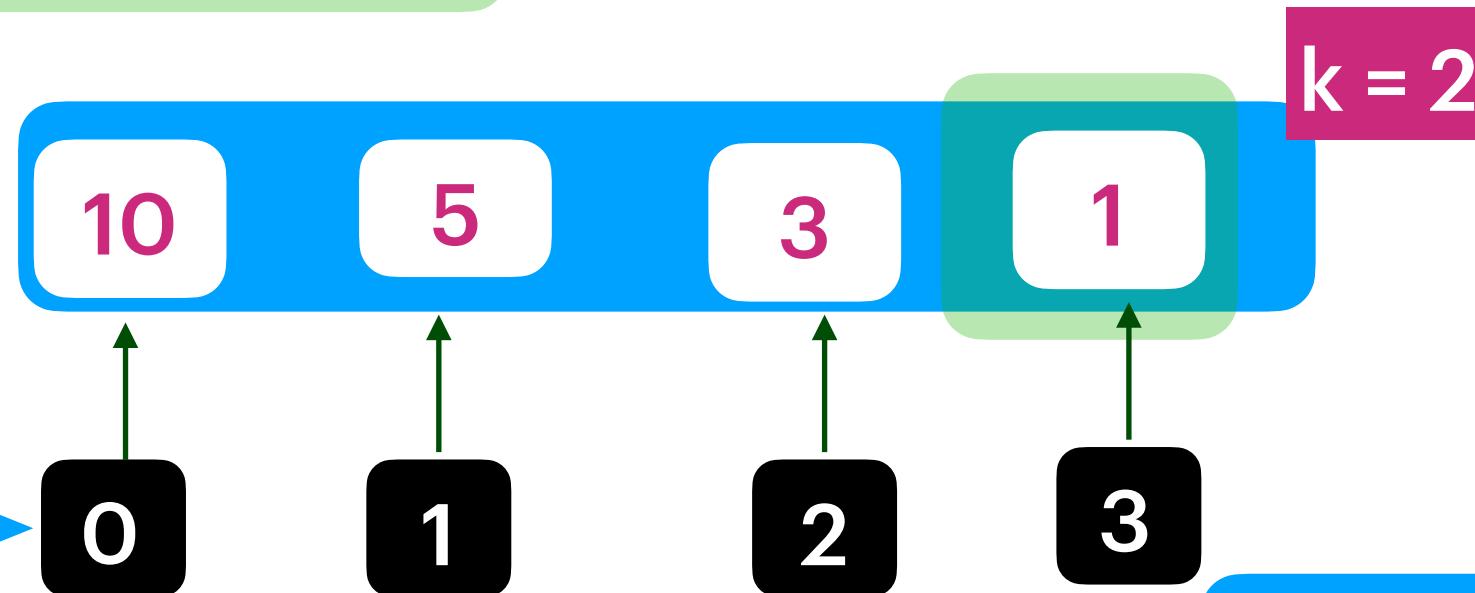
dummyTail → E6 → E5 → E4 <-dummyHead

**Algorithm:**

Always add new elements to the rear ,  
remove the fro when the front element is out of the window.



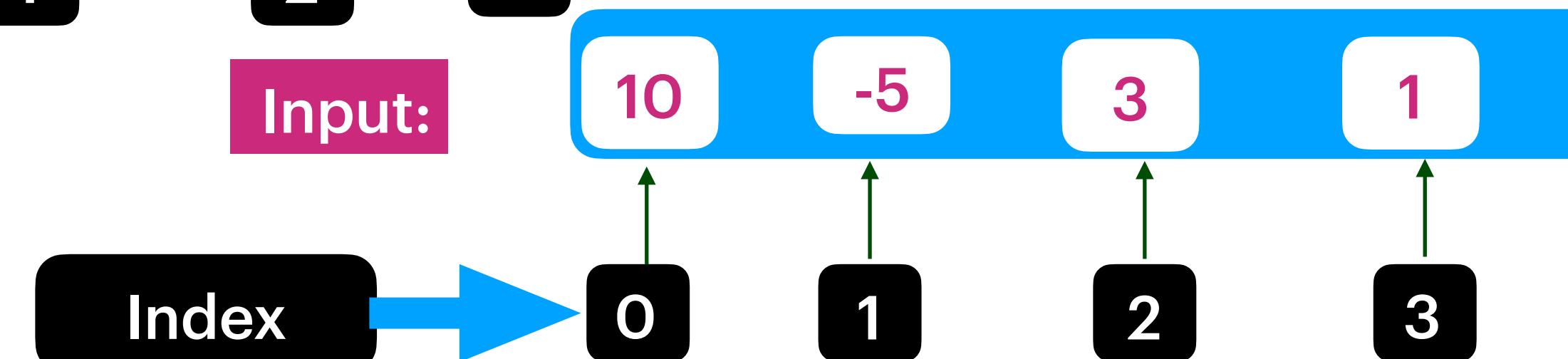
**Input:**



10 :  
 $10+5 = 15 \Rightarrow \text{Max}(10,15) = 15$   
 $15+3 = 18 \Rightarrow \text{Max}(15,18) = 18$   
 $18+1 = 19 = \text{Max}(18,19) = 19$

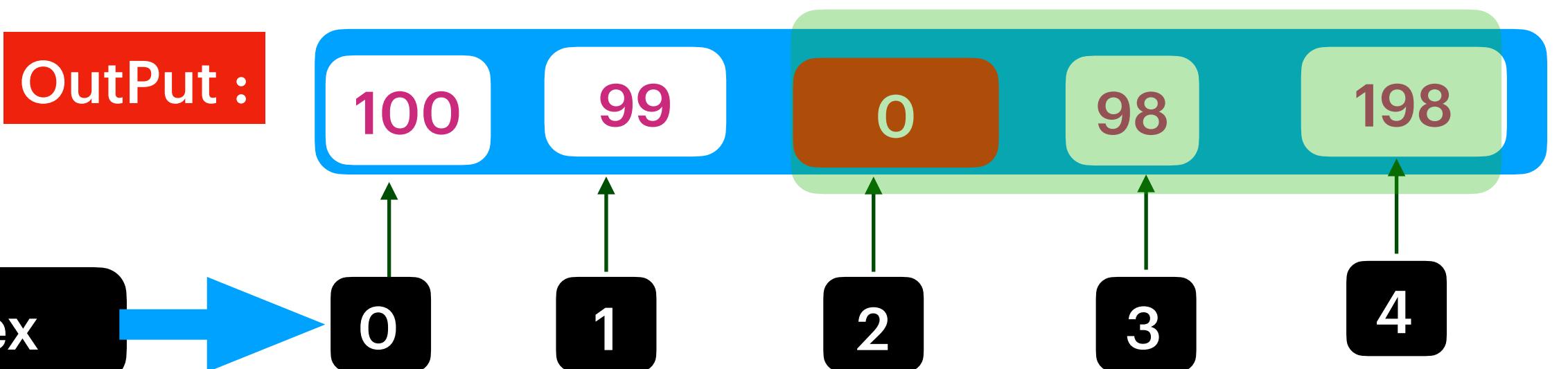
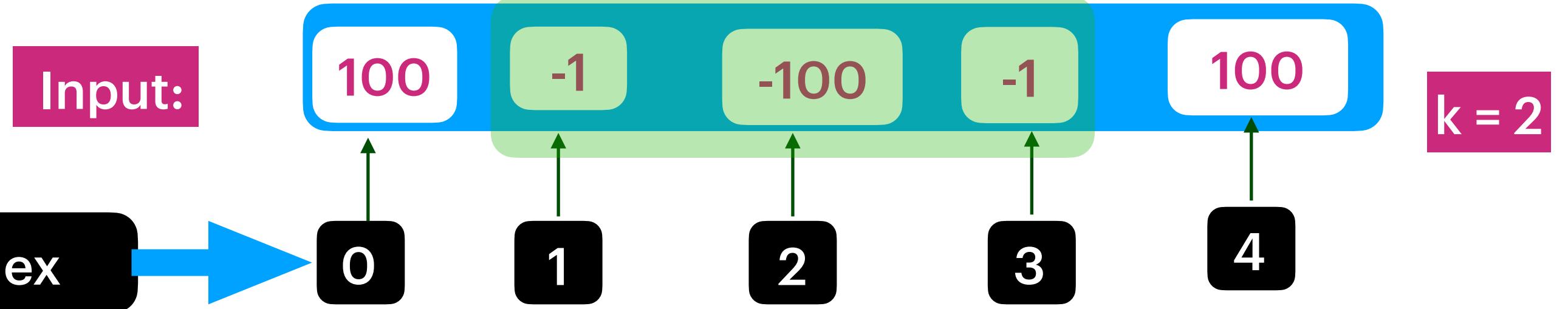
**Index**

**Input:**



**Input:**

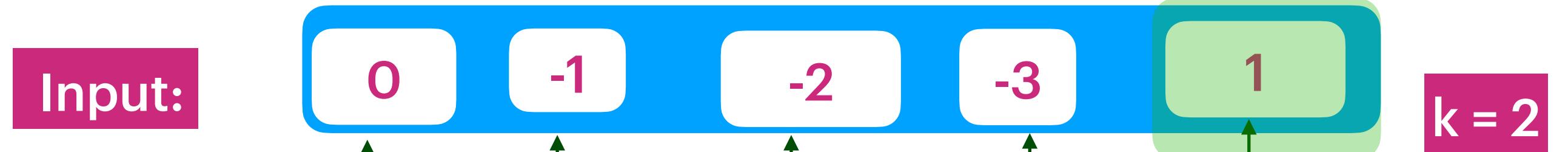
10 :  
 $10-5 = 5 \Rightarrow \text{Max}(10,5) = 10$   
 $10+3 = 13 \Rightarrow \text{Max}(13,10) = 13$   
 $13+1 = 14 \Rightarrow \text{Max}(13,14) = 14$



**10 :**  
 $10 - 5 = 5 \Rightarrow \text{Max}(10, 5) \Rightarrow 10$   
 $10 + 3 = 13 \Rightarrow \text{Max}(13, 10) \Rightarrow 13$   
 $13 + 1 = 14 \Rightarrow \text{Max}(13, 14) \Rightarrow 14$

**Deque :**

dummyTail → Index4 <— dummyHead



**Index** → 0    1    2    3    4



**Index** → 0    1    2    3    4

**Dequeue :**

dummyTail → Index4 <— dummyHead

**Input:** nums = [10,-5,-2,4,0,3] k = 3

**Output:** 17

**Explanation:** You can choose your jumps forming the subsequence [10,4,3]. The sum is 17.

## Zigzag Iterator

Given two vectors of integers v1 and v2, implement an iterator to return their elements alternately.

Implement the ZigzagIterator class:

`ZigzagIterator(List<int> v1, List<int> v2)` initializes the object with the two vectors v1 and v2.

`boolean hasNext()` returns true if the iterator still has elements, and false otherwise.

`int next()` returns the current element of the iterator and moves the iterator to the next element.

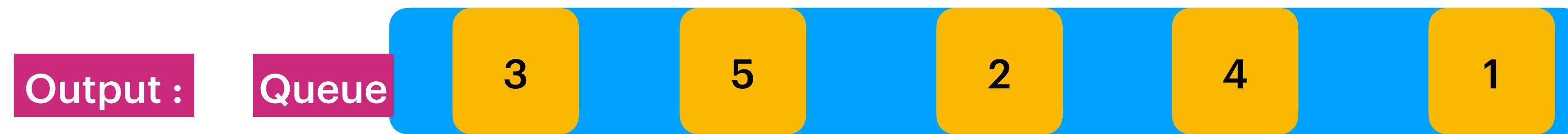
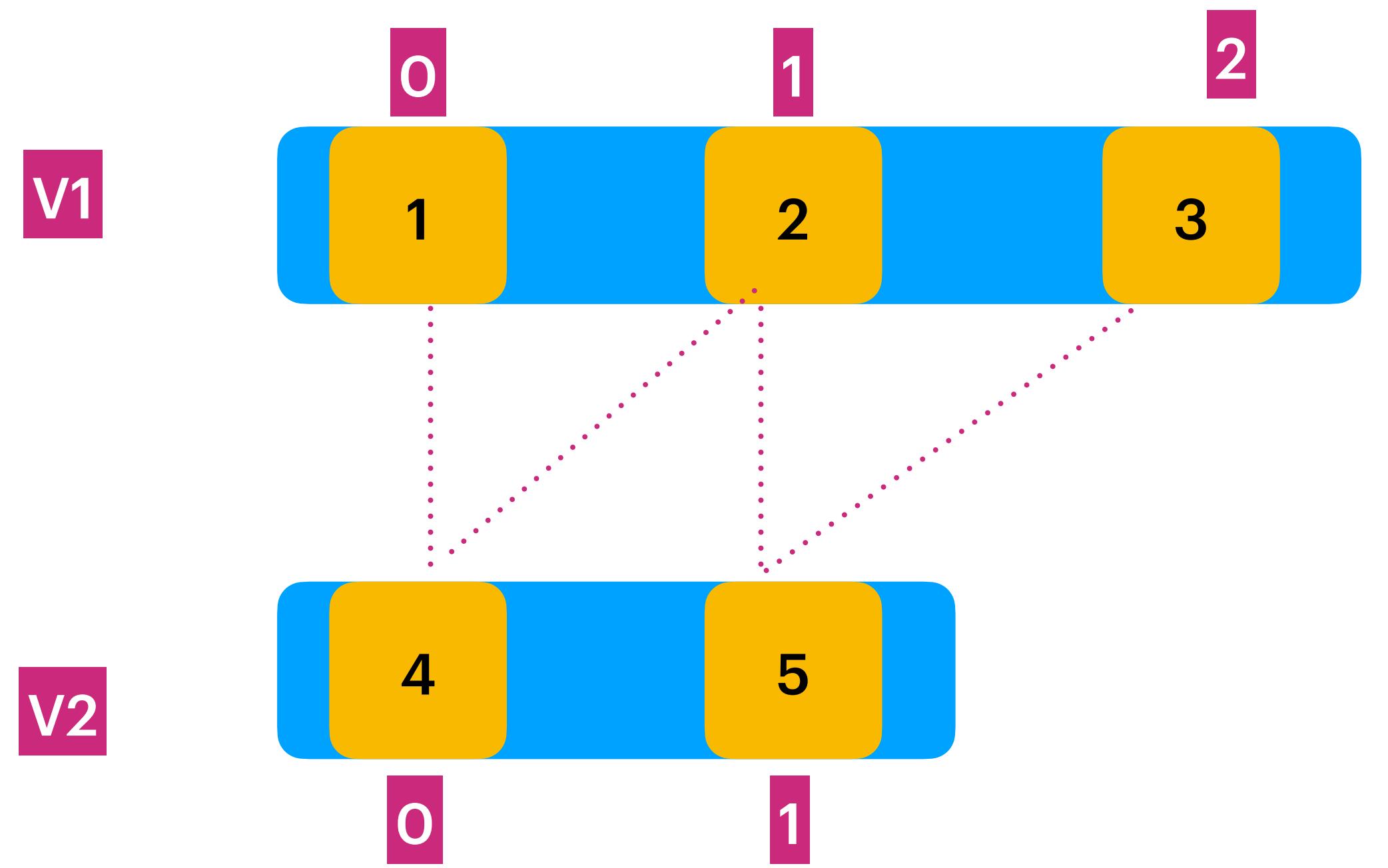
Input: v1 = [1,2], v2 = [3,4,5,6]

Output: [1,3,2,4,5,6]

Explanation: By calling `next` repeatedly until `hasNext` returns false,  
the order of elements returned by `next` should be: [1,3,2,4,5,6].

Input: v1 = [1], v2 = []

Output: [1]



## Time Needed to Buy Tickets

There are  $n$  people in a line queuing to buy tickets, where the 0th person is at the front of the line and the  $(n - 1)$ th person is at the back of the line.

You are given a 0-indexed integer array `tickets` of length  $n$  where the number of tickets that the  $i$ th person would like to buy is `tickets[i]`.

Each person takes exactly 1 second to buy a ticket. A person can only buy 1 ticket at a time and has to go back to the end of the line (which happens instantaneously) in order to buy more tickets.

If a person does not have any tickets left to buy, the person will leave the line.

Return the time taken for the person at position  $k$  (0-indexed) to finish buying tickets.

**Input:** `tickets` = [2,3,2],  $k$  = 2

Output: 6

Explanation:

- In the first pass, everyone in the line buys a ticket and the line becomes [1, 2, 1].
  - In the second pass, everyone in the line buys a ticket and the line becomes [0, 1, 0].
- The person at position 2 has successfully bought 2 tickets and it took  $3 + 3 = 6$  seconds.

**Input:** `tickets` = [2,3,2],  $k$  = 2

Output: 6

Explanation:

- In the first pass, everyone in the line buys a ticket and the line becomes [1, 2, 1].
  - In the second pass, everyone in the line buys a ticket and the line becomes [0, 1, 0].
- The person at position 2 has successfully bought 2 tickets and it took  $3 + 3 = 6$  seconds.

`tickets = [2[0],3[1],2[2]] k = 2`

`[2[0],3[1],2[2]] 0sec 2[2] -> 3[1] -> 2[0] [Front] : counter = 0`

`[3[1],2[2],1[0]] 1Sec 1[0] -> 2[2] -> 3[1] [Front] : counter = 1`

`[2[2],1[0],2[1]] 2Sec 2[1] -> 1[0] -> 2[2] [Front] : counter = 2`

`[1[0],2[1],1[2]] 3Sec 1[2] -> 2[1] -> 1[0] [Front] : counter = 3`

`[2[1],1[2],0[0](X)] 4Sec 0[0](X) don't add to deque  
as the value zero`

`[2[1],1[2]] 4Sec 1[2] -> 2[1] [Front] : counter = 4`

`[1[2],1[1]] 5Sec 1[1] -> 1[2] [Front] : counter = 5`

`[1[1],0[2] (X)] 6Sec 0[2] (X) as k == 2 & value = 0 return counter : 6  
1[1] [Front] : counter = 6`

`tickets = [2[0],3[1],2[2]] k = 1`

`[2[0],3[1],2[2]] 0sec`

`[3[1],2[2],1[0]] 1Sec`

`[2[2],1[0],2[1]] 2Sec`

`[1[0],2[1],1[2]] 3Sec`

`[2[1],1[2],0[0](X)] 4Sec`

`[2[1],1[2]] 4Sec`

`[1[2],1[1]] 5Sec`

`[1[1],0[2] (X)] 6Sec`

`[1[1]] 6Sec`

`[0[1] X] 7Sec`

`[] 7Sec`

## Number of Students Unable to Eat Lunch

The school cafeteria offers circular and square sandwiches at lunch break, referred to by numbers 0 and 1 respectively. All students stand in a queue. Each student either prefers square or circular sandwiches.

The number of sandwiches in the cafeteria is equal to the number of students. The sandwiches are placed in a stack. At each step:

If the student at the front of the queue prefers the sandwich on the top of the stack, they will take it and leave the queue.

Otherwise, they will leave it and go to the queue's end

This continues until none of the queue students want to take the top sandwich and are thus unable to eat.

You are given two integer arrays `students` and `sandwiches` where `sandwiches[i]` is the type of the  $i$ th sandwich in the stack ( $i = 0$  is the top of the stack) and `students[j]` is the preference of the  $j$ th student in the initial queue ( $j = 0$  is the front of the queue). Return the number of students that are unable to eat.

Input: `students` = [1,1,0,0], `sandwiches` = [0,1,0,1]

Output: 0

Explanation:

- Front student leaves the top sandwich and returns to the end of the line making `students` = [1,0,0,1].
- Front student leaves the top sandwich and returns to the end of the line making `students` = [0,0,1,1].
- Front student takes the top sandwich and leaves the line making `students` = [0,1,1] and `sandwiches` = [1,0,1].
  - Front student leaves the top sandwich and returns to the end of the line making `students` = [1,1,0].
  - Front student takes the top sandwich and leaves the line making `students` = [1,0] and `sandwiches` = [0,1].
    - Front student leaves the top sandwich and returns to the end of the line making `students` = [0,1].
    - Front student takes the top sandwich and leaves the line making `students` = [1] and `sandwiches` = [1].
    - Front student takes the top sandwich and leaves the line making `students` = [] and `sandwiches` = [].

Hence all students are able to eat.

Input: `students` = [1,1,1,0,0,1], `sandwiches` = [1,0,0,0,1,1]

Output: 3

Input: students [1,1,0,0], sandwiches = [0,1,0,1]

0 - Circuler

1 - Square

students[1,1,0,0] 0 -> 0 -> 1 -> 1(front) sandwiches[0,1,0,1] 1 -> 0 -> 1 -> 0 (front)

students[1,0,0,1] -> 1 -> 0 -> 0 -> 1 (front) sandwiches[0<-1<-0<-1]

students[0,0,1,1] -> -> 1 -> 1 -> 0 -> 0 sandwiches[0,1,0,1]

students[0,1,1] -> 1 -> 1 -> 0 sandwiches[1,0,1]

students[1,1,0] -> 0 -> 1 -> 1 sandwiches[1,0,1]

students[1,0] -> 0 -> 1 sandwiches[0,1]

students[0,1] -> 1 -> 0 sandwiches[0,1]

students[1] -> 1 sandwiches[1]

students[] -> sandwiches[]

output: 0

\*\*\*\*\*

**students [1,1,1,0,0,1], sandwiches[1,0,0,0,1,1]**

**students [1,1,0,0,1], sandwiches[0,0,0,1,1]**

**students [1,0,0,1,1], sandwiches[0,0,0,1,1]**

**students [0,0,1,1,1], sandwiches[0,0,0,1,1]**

**students [0,1,1,1], sandwiches[0,0,1,1]**

**counter = 0**

**students [1,1,1], sandwiches[0,1,1]**

**students [1,1,1], sandwiches[0,1,1] : counter = 1**

**students [1,1,1], sandwiches[0,1,1] : counter = 2**

**students [1,1,1], sandwiches[0,1,1] : counter= 3**

**output : 3**

**TimeComplexity O(2n)**

**4 <-> 4**

**n**

**[1]+students[3] <-> sandwiches[3]**

**[2]+ 2+1 <-> 3**

**[3] + 1 +2 <-> 3**

**[4] + 2 <-> 2**

**[5] + 1+1 <-> 2**

**[5+1] + 1 <-> 1**

## Design Hit Counter

Design a hit counter which counts the number of hits received in the past 5 minutes (i.e., the past 300 seconds).

Your system should accept a timestamp parameter (in seconds granularity), and you may assume that calls are being made to the system in chronological order (i.e., timestamp is monotonically increasing). Several hits may arrive roughly at the same time.

Implement the HitCounter class:

**HitCounter()** Initializes the object of the hit counter system.  
**void hit(int timestamp)** Records a hit that happened at timestamp (in seconds). Several hits may happen at the same timestamp.  
**int getHits(int timestamp)** Returns the number of hits in the past 5 minutes from timestamp (i.e., the past 300 seconds).

### Input

```
["HitCounter", "hit", "hit", "hit", "getHits", "hit", "getHits", "getHits"]
[], [1], [2], [3], [4], [300], [300], [301]
```

### Output

```
[null, null, null, null, 3, null, 4, 3]
```

### Explanation

```
HitCounter hitCounter = new HitCounter();
    hitCounter.hit(1);    // hit at timestamp 1.
    hitCounter.hit(2);    // hit at timestamp 2.
    hitCounter.hit(3);    // hit at timestamp 3.
    hitCounter.getHits(4); // get hits at timestamp 4, return 3.
    hitCounter.hit(300);  // hit at timestamp 300.
    hitCounter.getHits(300); // get hits at timestamp 300, return 4.
    hitCounter.getHits(301); // get hits at timestamp 301, return 3.
```

**1sec -> 2 sec -> 3sec -> 4sec**

**gitHits(300)// output: 4**  $5*60 = 300\text{sec} : 4$

$300-4 = 296\text{sec}$

$300-3 = 297\text{Sec}$

$300-2 = 298\text{Sec}$

$300-1 = 299\text{Sec}$

$4 \rightarrow 3 \rightarrow 2 \rightarrow 1$

**gitHits(301)// output: 3**

$301-4 = 297\text{sec}$

$301-3 = 298\text{Sec}$

$301-2 = 299\text{Sec}$

$301-1 = 300 (X)$

$4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \Rightarrow 301 - 1 = 300 \geq 300$

**gitHits(303)// output: 1**

$303-4 = 299\text{sec}$

$303-3 = 300\text{Sec}$

$303-2 = 301\text{Sec}$

$303-1 = 302\text{Sec}$

$4 \Rightarrow 303-4 \ 299 \geq 300$

**1sec -> 2 sec -> 2 sec -> 2 sec -> 3sec -> 4sec**

**gitHits(300)// output: 6**  $5*60 = 300\text{sec} : 4$

$300-4 = 296\text{sec}$

$300-3 = 297\text{Sec}$

$300-2 = 298\text{Sec} (3)$

$300-1 = 299\text{Sec}$

**1sec -> 2 sec -> 2 sec -> 2 sec -> 3sec -> 4sec**

**gitHits(304)// output: 6**  $5*60 = 300\text{sec} : 4$

## Sliding Window Maximum

You are given an array of integers  $\text{nums}$ , there is a sliding window of size  $k$  which is moving from the very left of the array to the very right. You can only see the  $k$  numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

**Input:**  $\text{nums} = [1,3,-1,-3,5,3,6,7]$ ,  $k = 3$

**Output:**  $[3,3,5,5,6,7]$

Explanation:

Window position	Max
-----	-----

[1 3 -1] -3 5 3 6 7	<b>3</b>
---------------------	----------

1 [3 -1 -3] 5 3 6 7	<b>3</b>
---------------------	----------

1 3 [-1 -3 5] 3 6 7	<b>5</b>
---------------------	----------

1 3 -1 [-3 5 3] 6 7	<b>5</b>
---------------------	----------

1 3 -1 -3 [5 3 6] 7	<b>6</b>
---------------------	----------

1 3 -1 -3 5 [3 6 7]	<b>7</b>
---------------------	----------

**Input:**  $\text{nums} = [1,-1]$ ,  $k = 1$

**Output:**  $[1,-1]$

**Input:**  $\text{nums} = [9,11]$ ,  $k = 2$

**Output:**  $[11]$

[1,3,-1,-3,5,3,6,7] k = 3

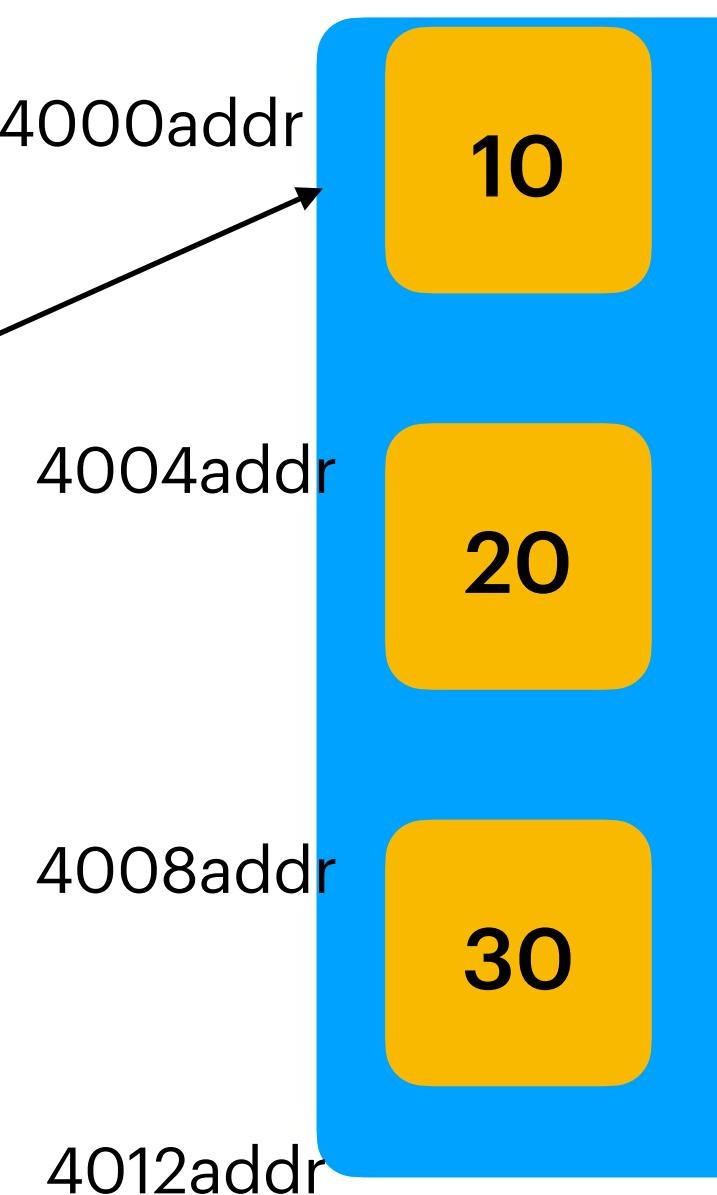
1,3,-1 => 3  
3, -1, -5 => 3  
-1, -3, 5 => 5  
-3, 5, 3 => 5  
5, 3, 6 => 6  
3, 6, 7=> 7

[1,3,-1,-3,5,3,6,7] k = 3

int[] arr = {10,20,30}

arr[0] = 4000addr  
+0\*4 = 4000addr

arr[2] = 4000 + 2\*4 = 4008



[1[0],0[1],-1[2],-3[3],-5[4]], 3

OUTPUT : 1,0,-1

-5 -> -3 -> -1

9 8 7 10 5 4 3 6 N= 4

~ 2N  
6 ->

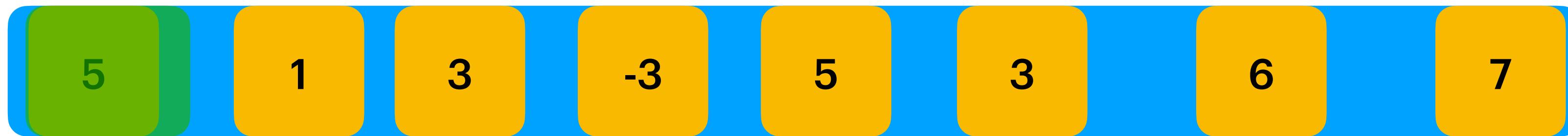
ADD 4 + REMOVE 3

ADD 4 + REMOVE 3

14

[1,3,-1,-3,5,3,6,7] k = 3

BruteForce Approach : O(nk)



output1 :: [3, 3, 5, 5, 6, 7]  
output2 :: [1, 0, -1]

Output : [3, 3, 5, 5, 6, 7]

N - K + 1

3 → 5

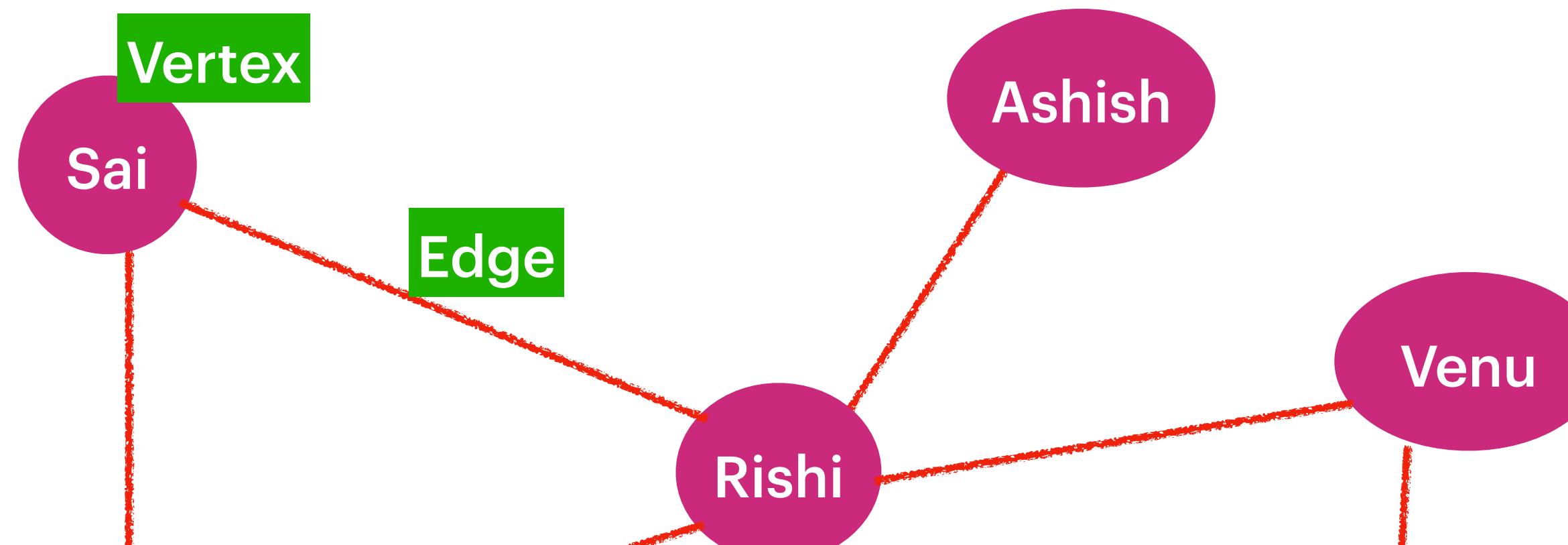
I:4. K = 3  
I-K = FRONT  
4-3 = 1

## Graph Terminology

### Graph

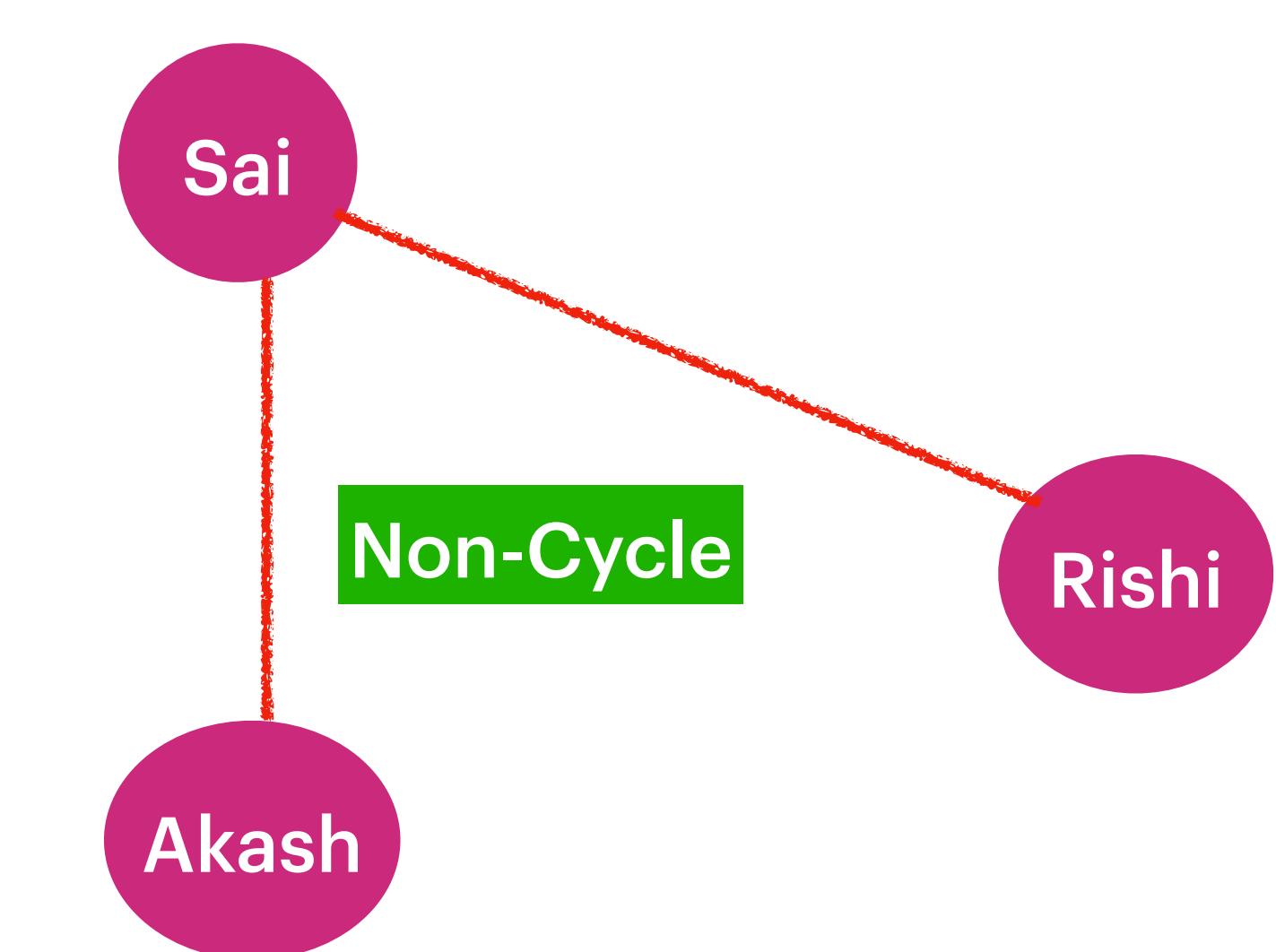
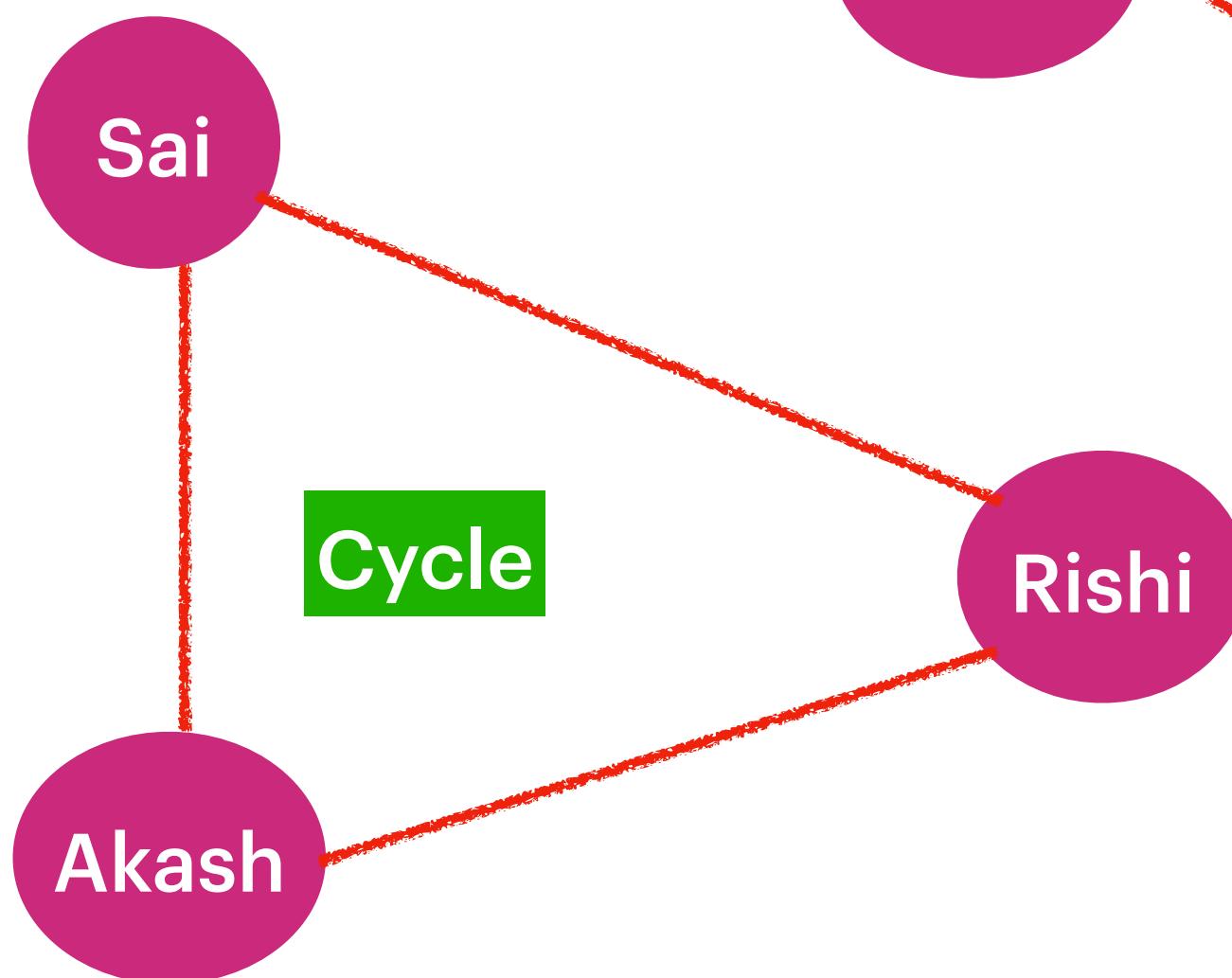
It's a way of connecting dot's.

#### Path



#### Connectivity

#### Degree of a Vertex

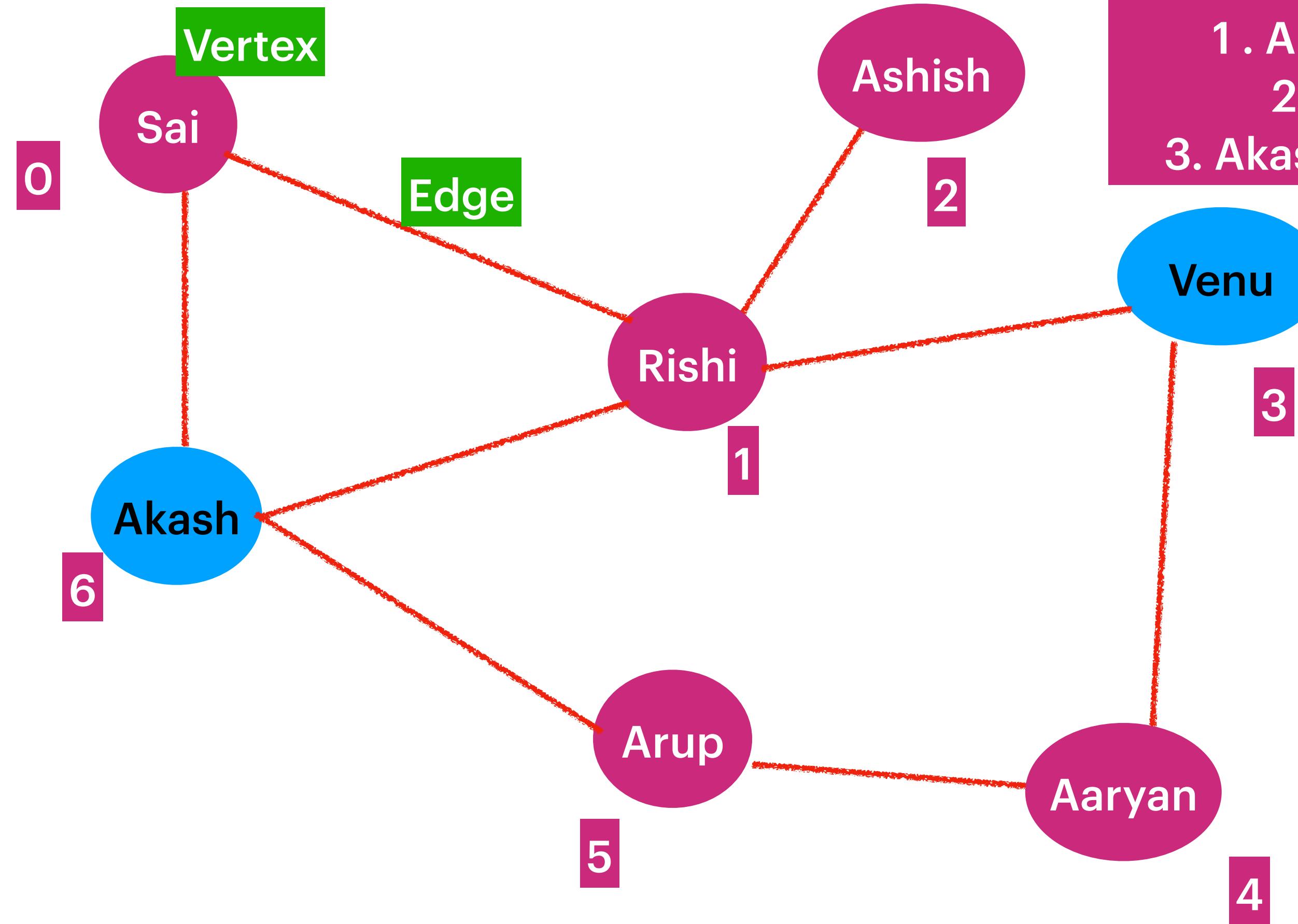


**Path :**

If there is connection exists,  
Path just talks about no.of between Source  
& Destination vertices.

**Graph**

It's a way of connecting dot's.



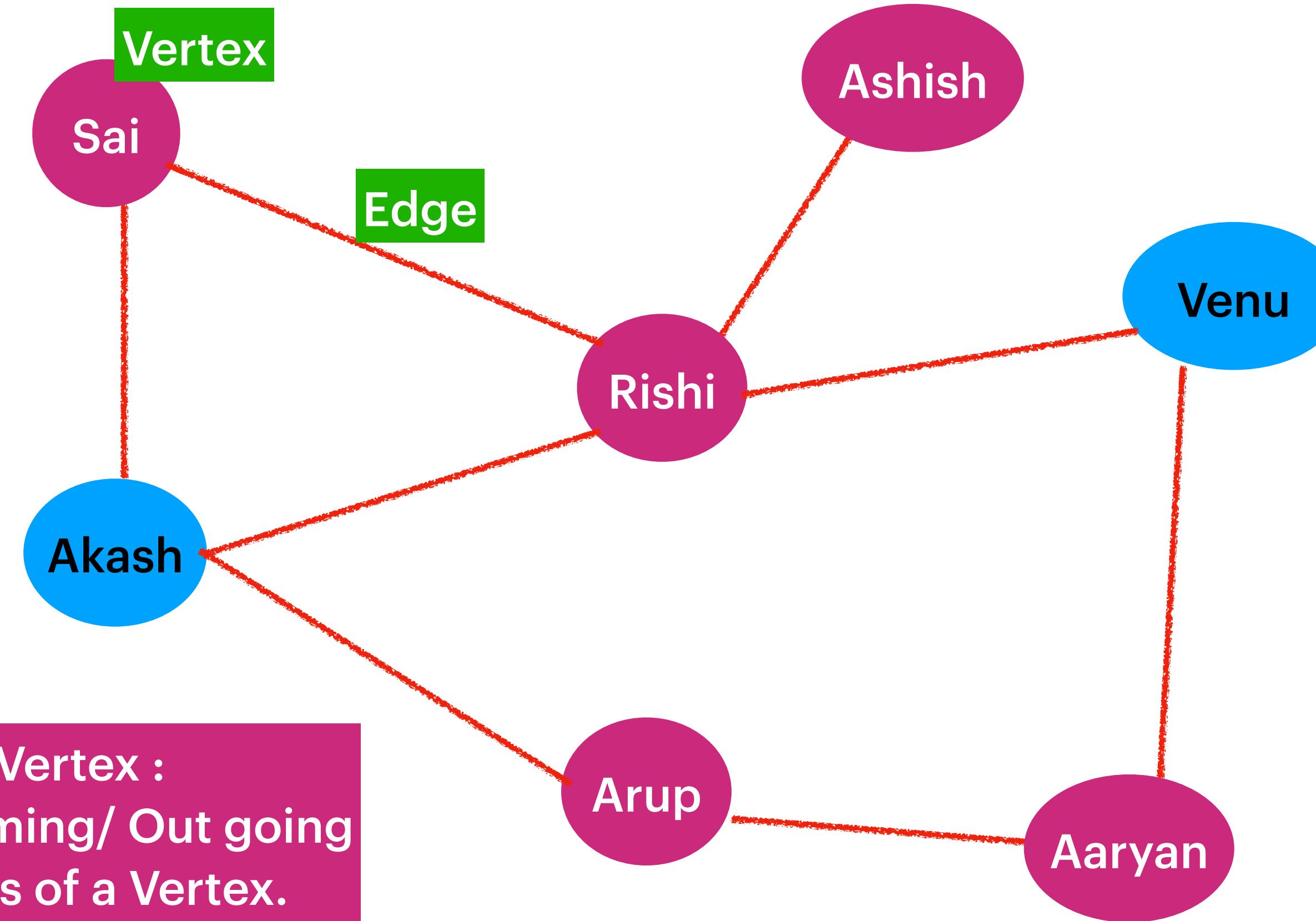
**Path between Akash <-> Venu**

Does path exists ? Yes

1. Akash — Sai — Rishi — Venu (3)
2. Akash — Rishi — Venu (2)
3. Akash — Arup — Aaryan — Venu (3)

## Bidirectional Graph

Here the connection exists in Both the ways.  
Source  $\longleftrightarrow$  Destination



**Degree Of Vertex :**  
Talks about Incoming/ Out going  
Connections of a Vertex.

**DegreeOf(Aakash) :**  
In-Degree =>3  
Out-Degree =>3

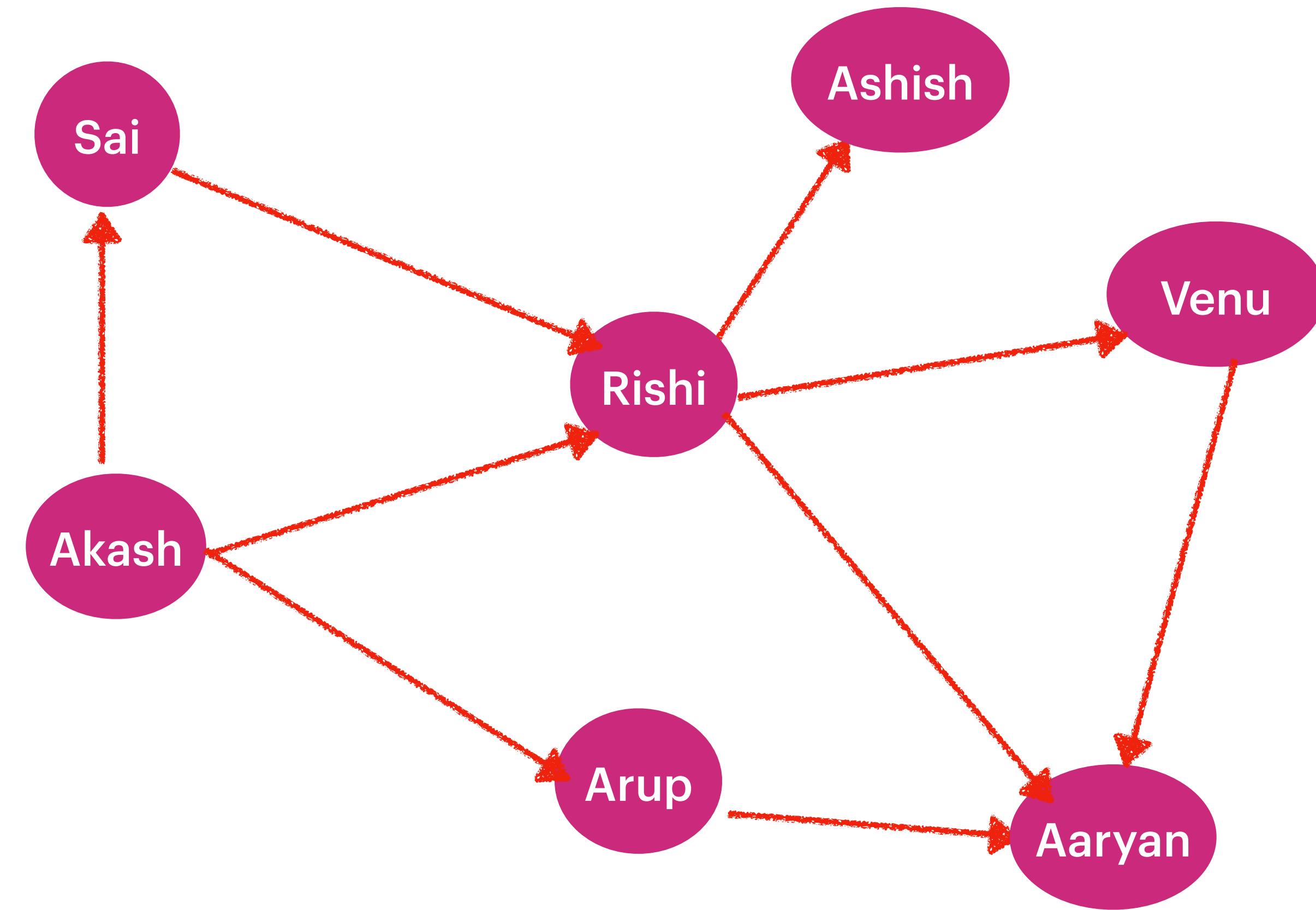
## Directed Graph

In-Degree (Sai) : 1

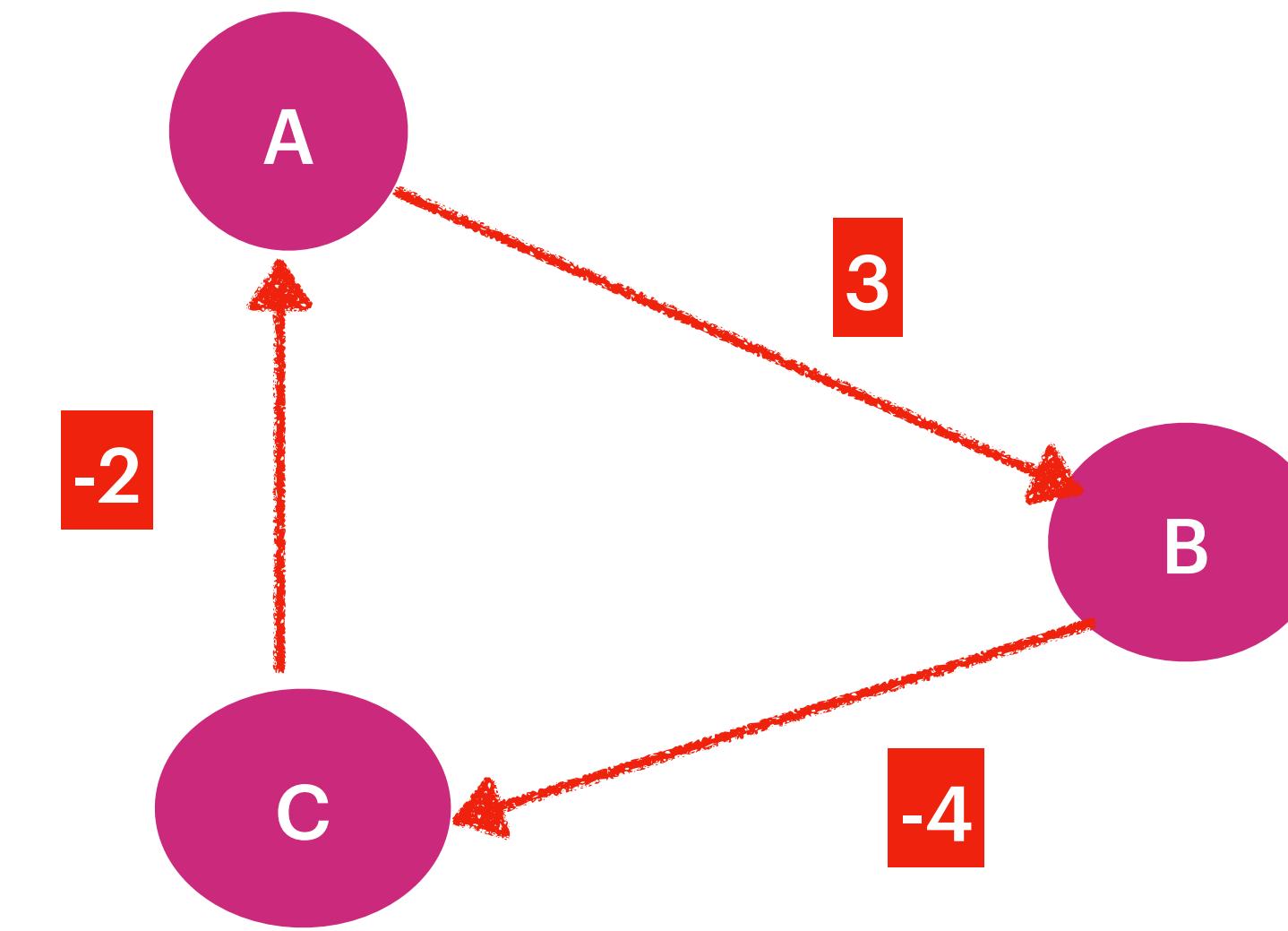
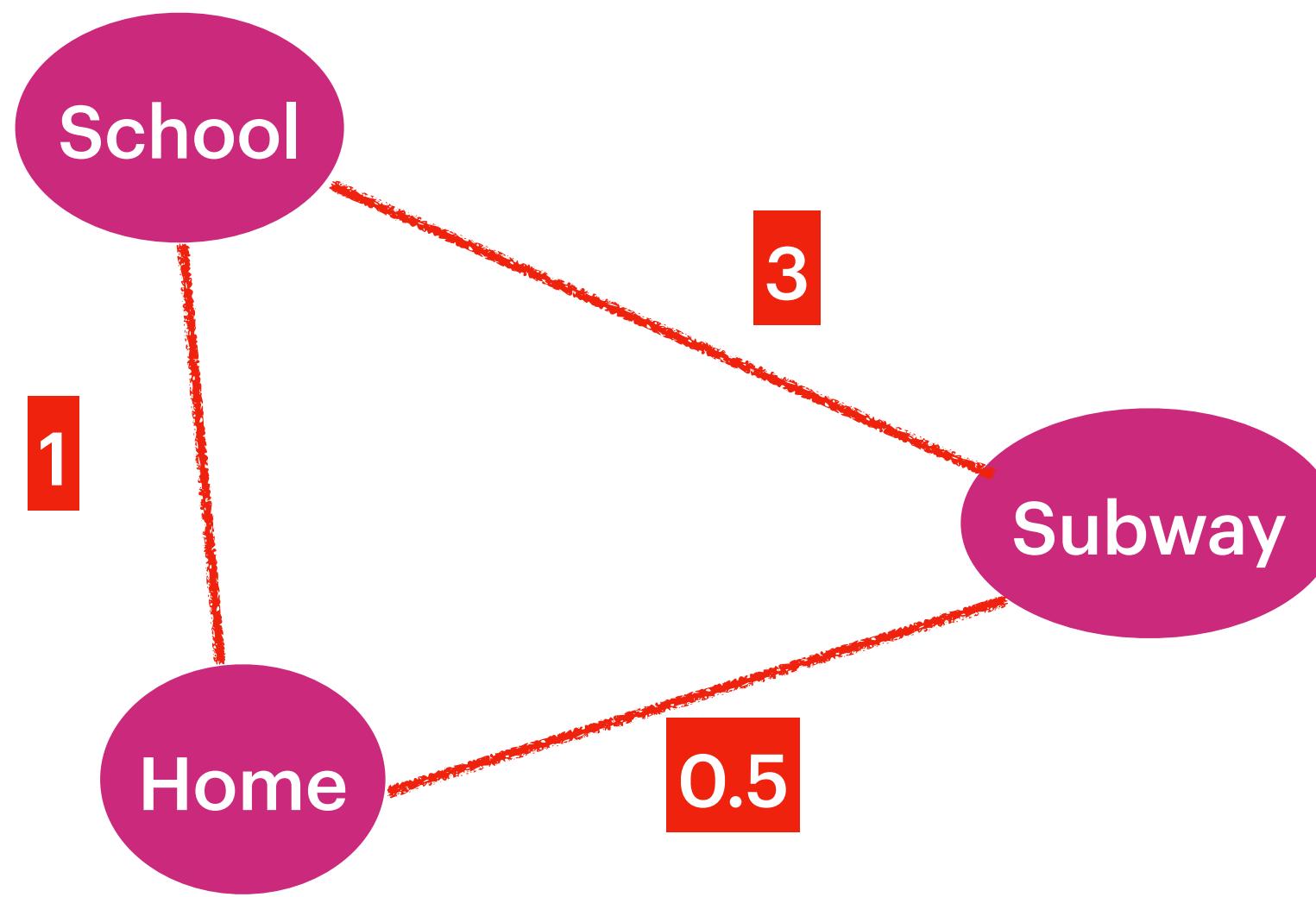
Out-Degree (Sai) : 1

In-Degree (Rishi) : 2

Out-Degree (Rishi) : 3

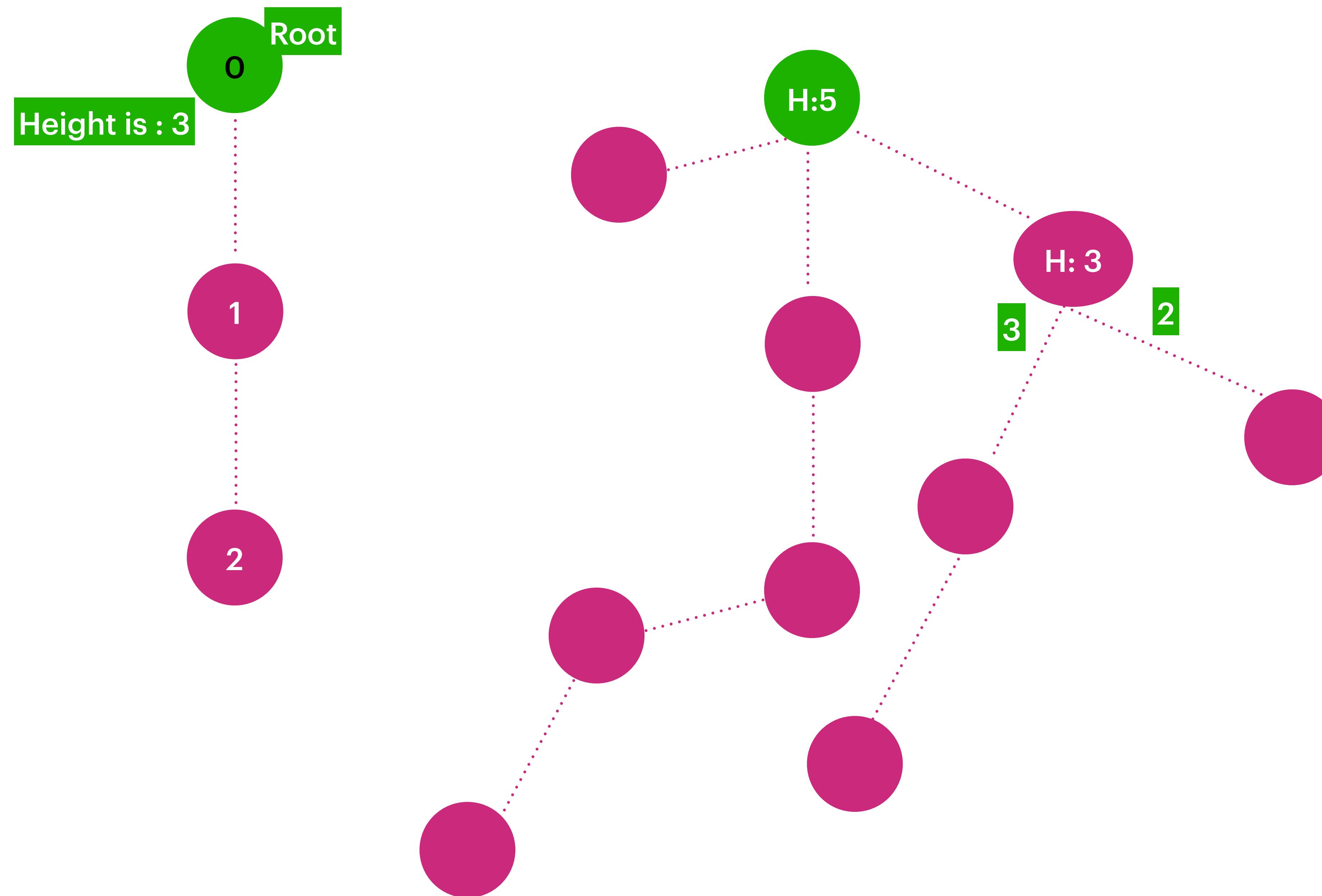


## Weighted Graph

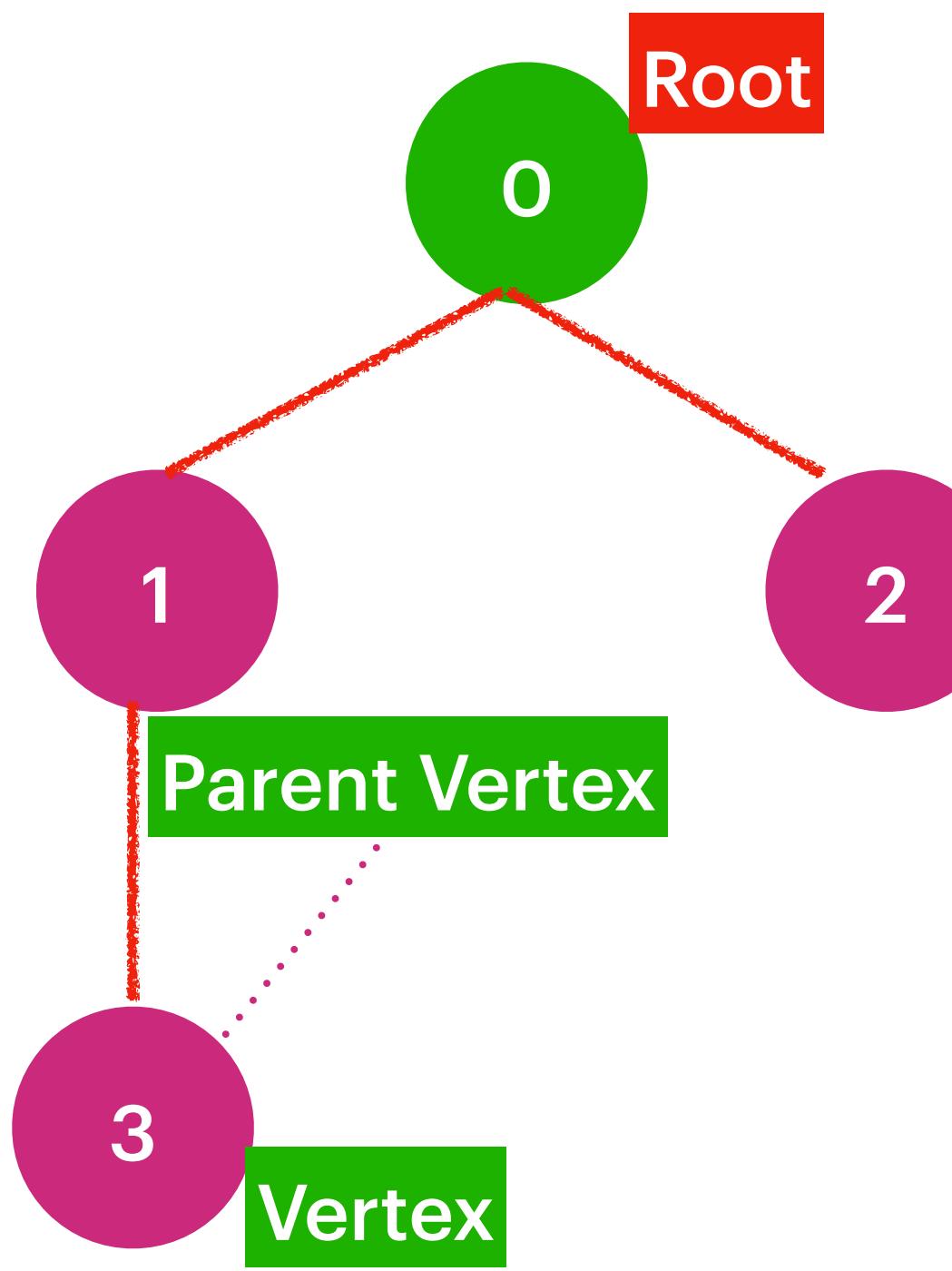


**Negative Weight Cycle =**  
 $A \rightarrow B \rightarrow C = 3 + (-4) + (-2) = -3$

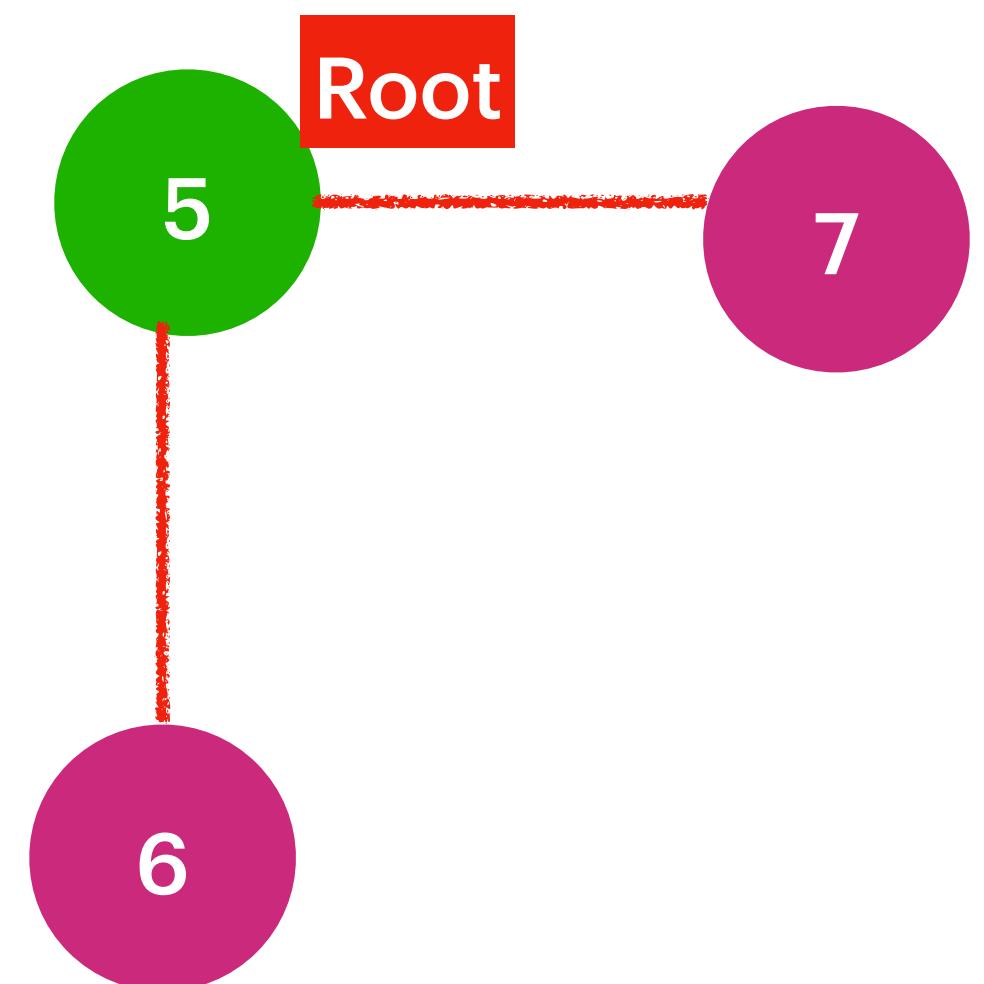
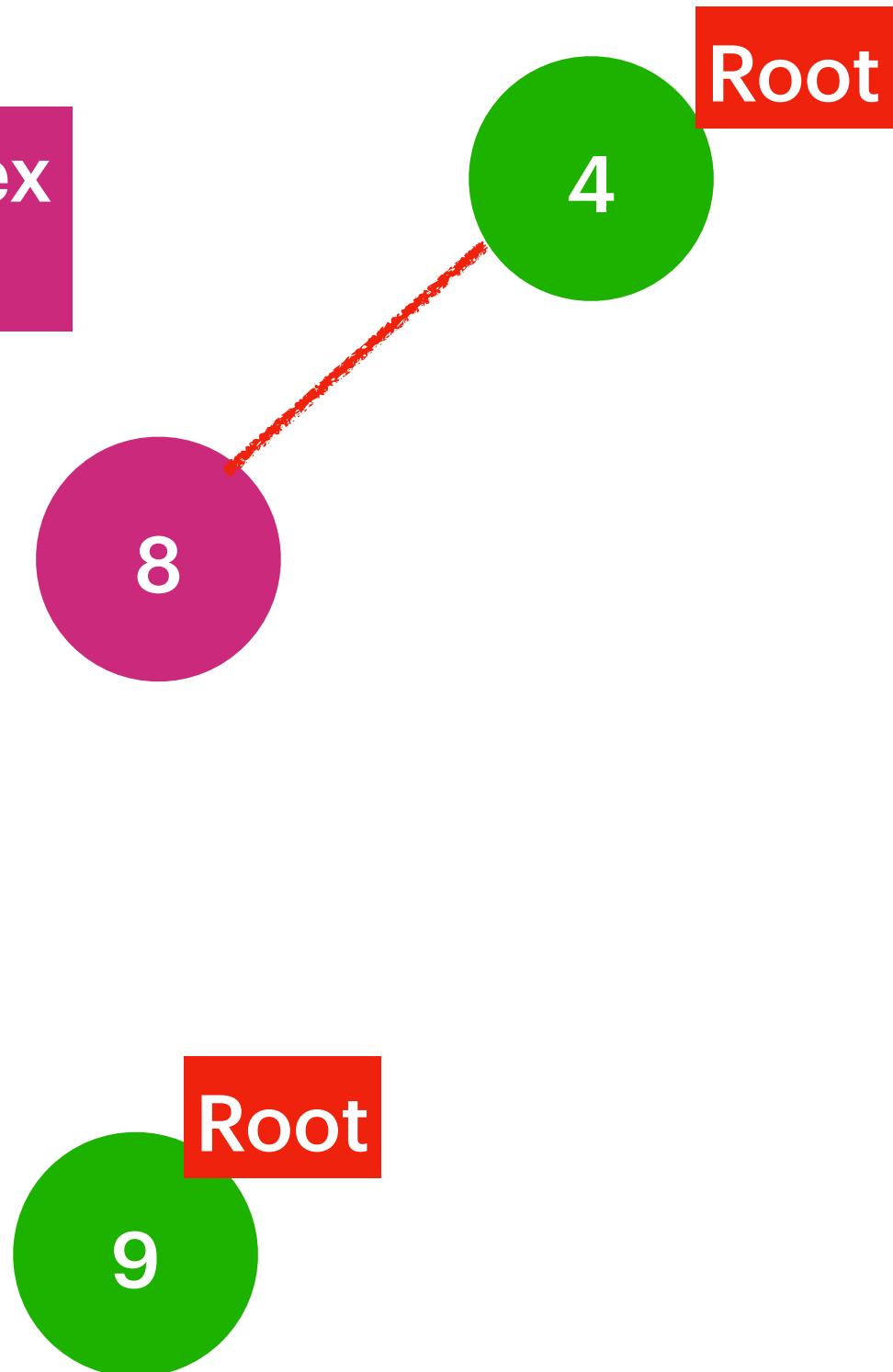
## Height Of The Graph

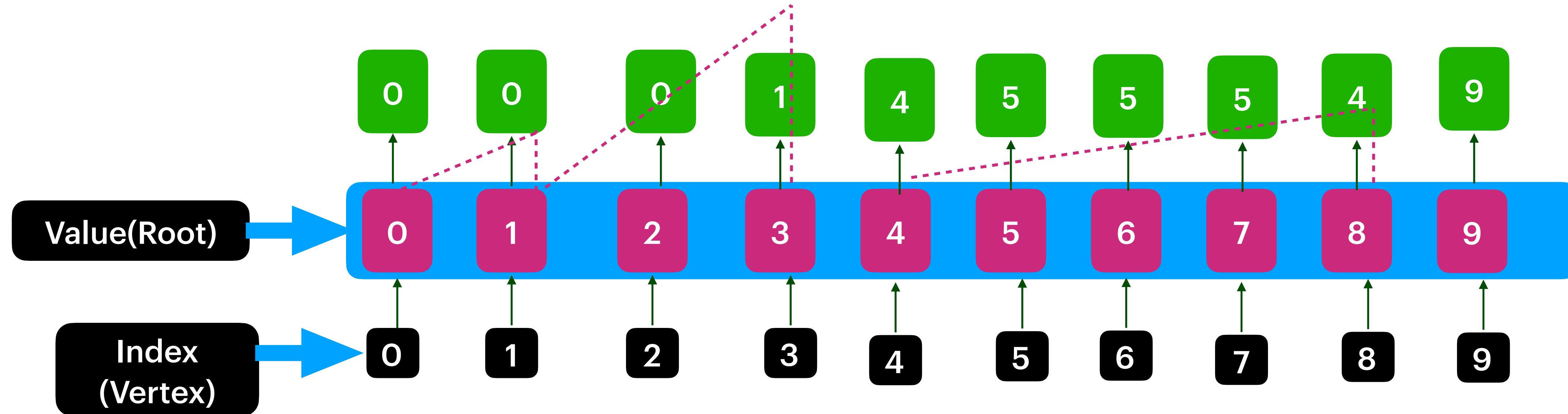


## Disjoint Set



How do we say that vertex  
are connected ?



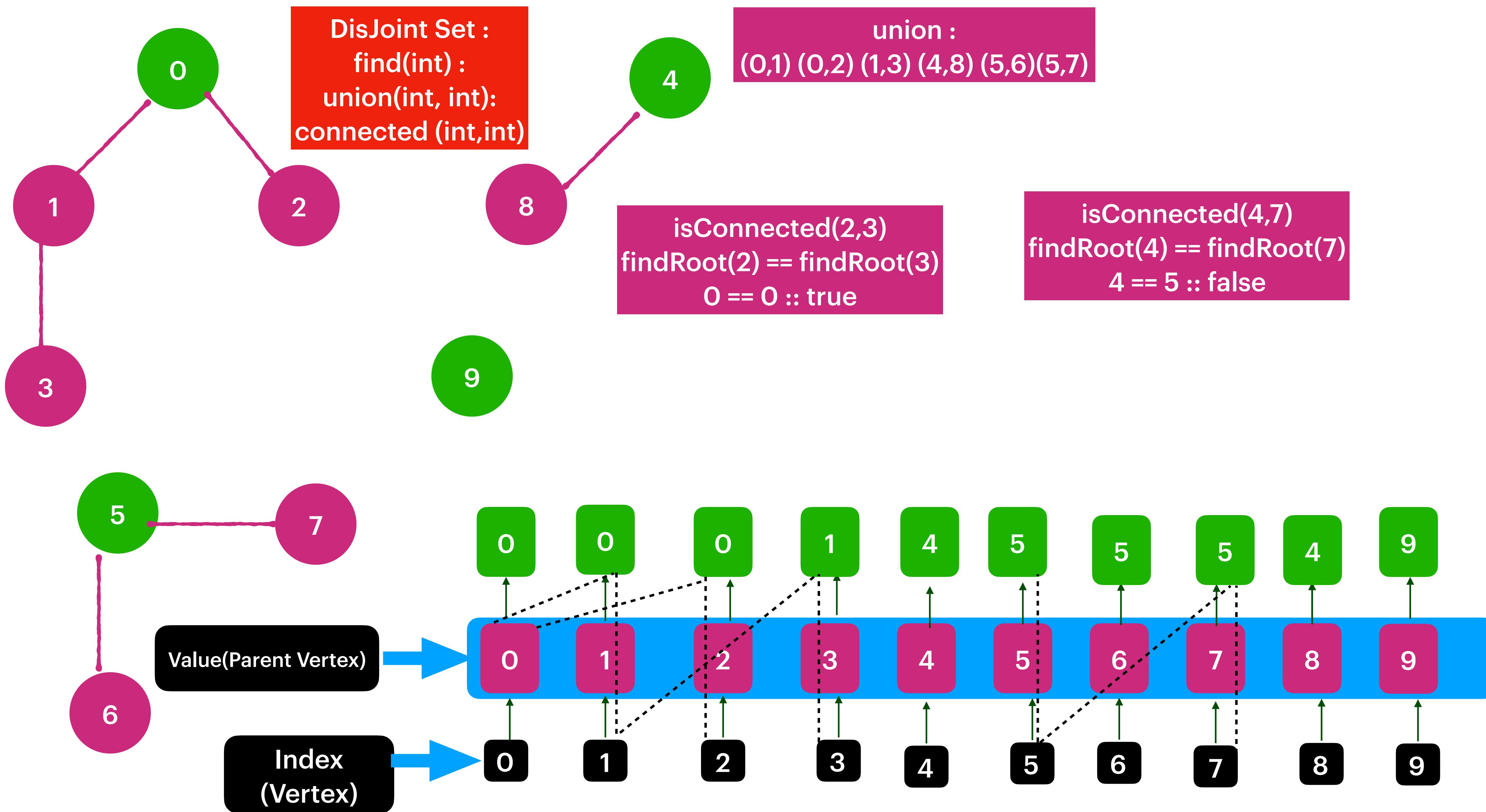


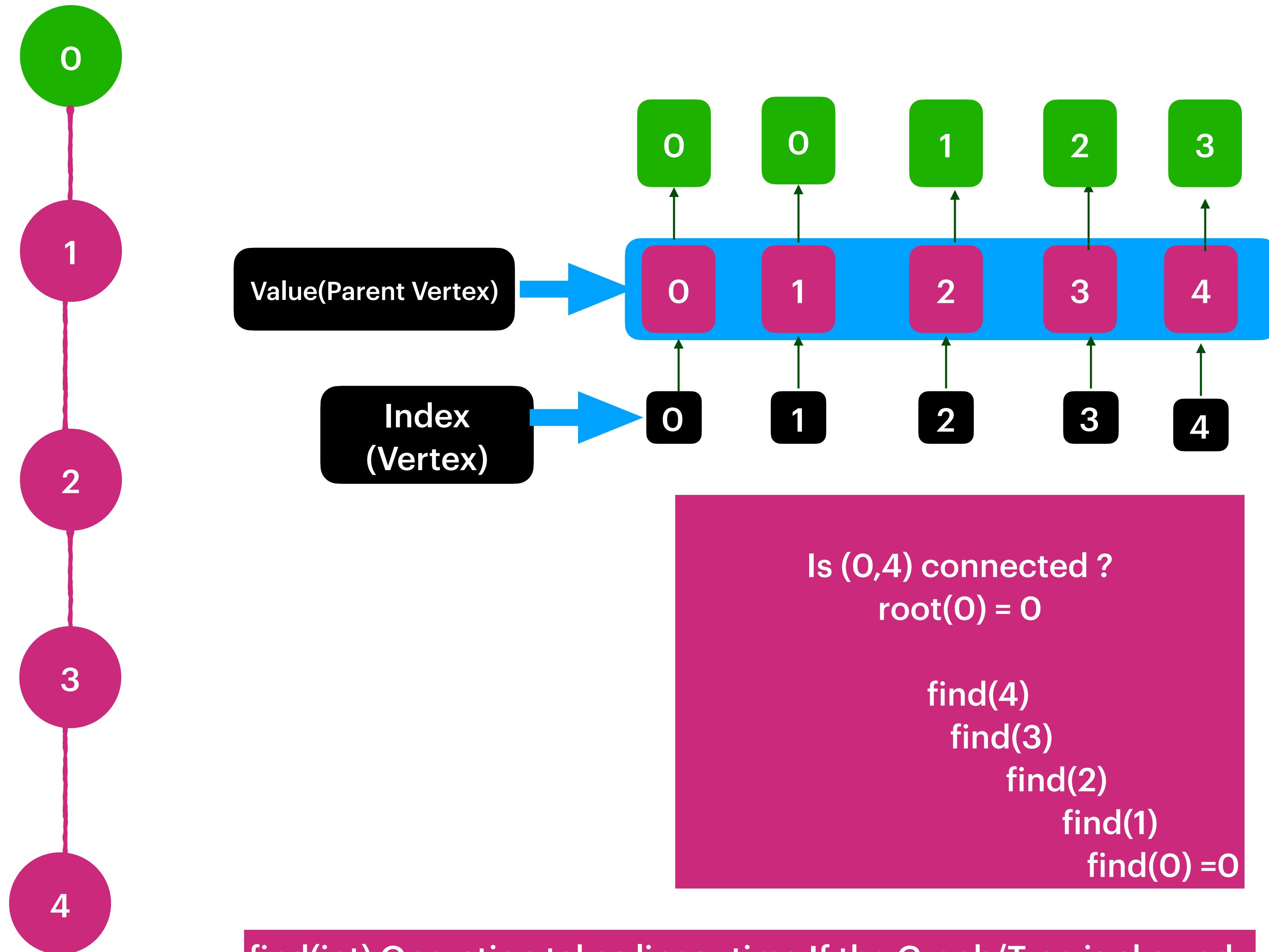
(0,1) , (0,2) , (1,3) , (4,8), (5,6), (5,7)

Connected(1,3)=  
root(1) == root(3)  
0 == 0 = true

connected(1,8)  
root(1) == root(8)  
0 == 4 (false)

connected(5,7)  
5 == 5 : true



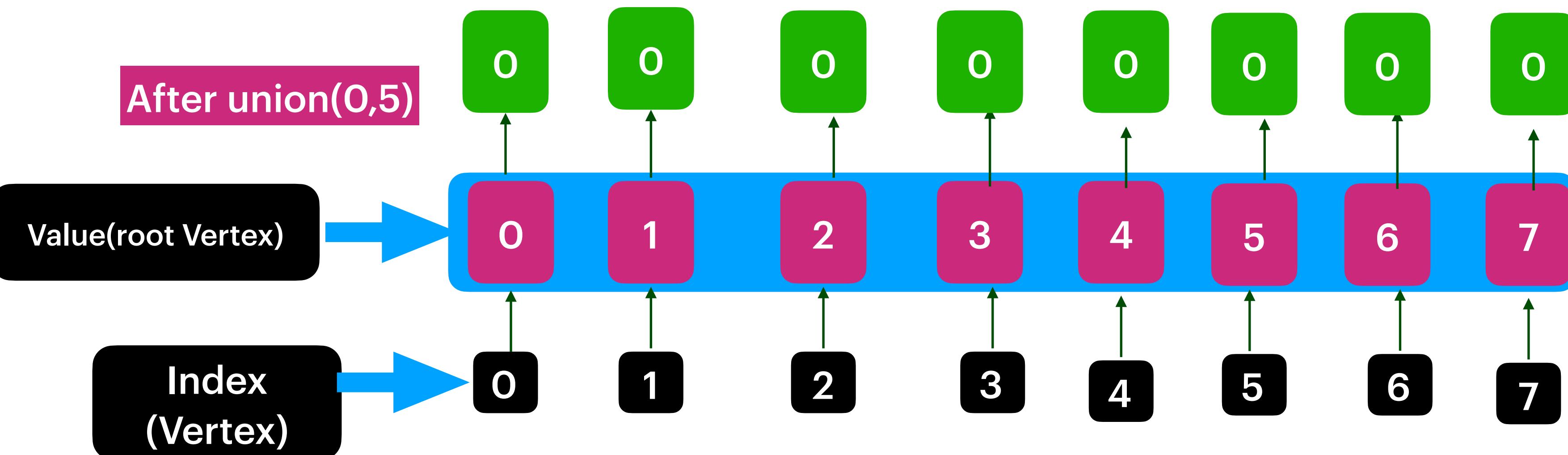
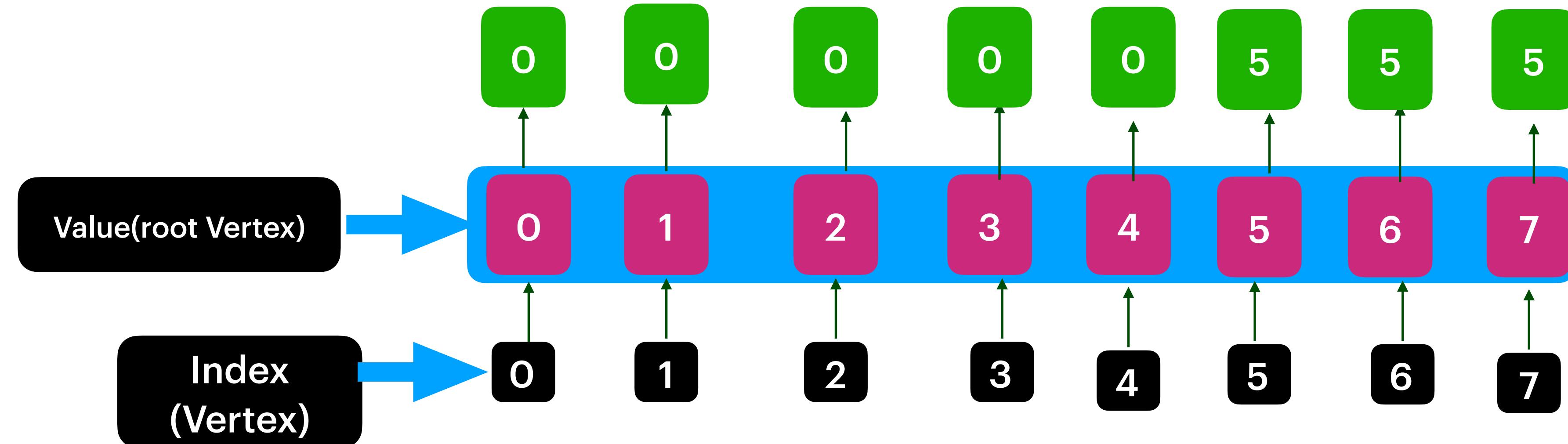
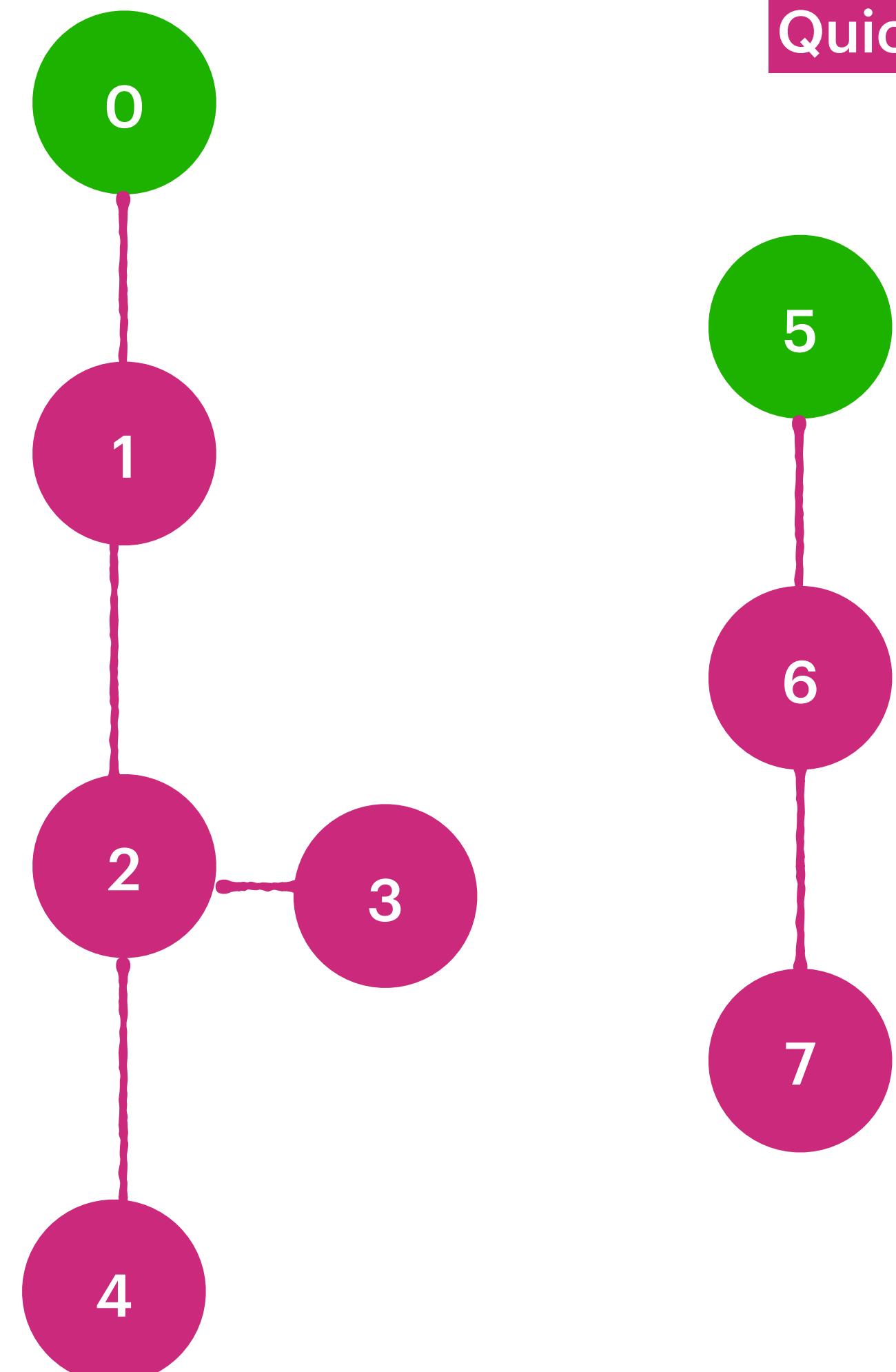


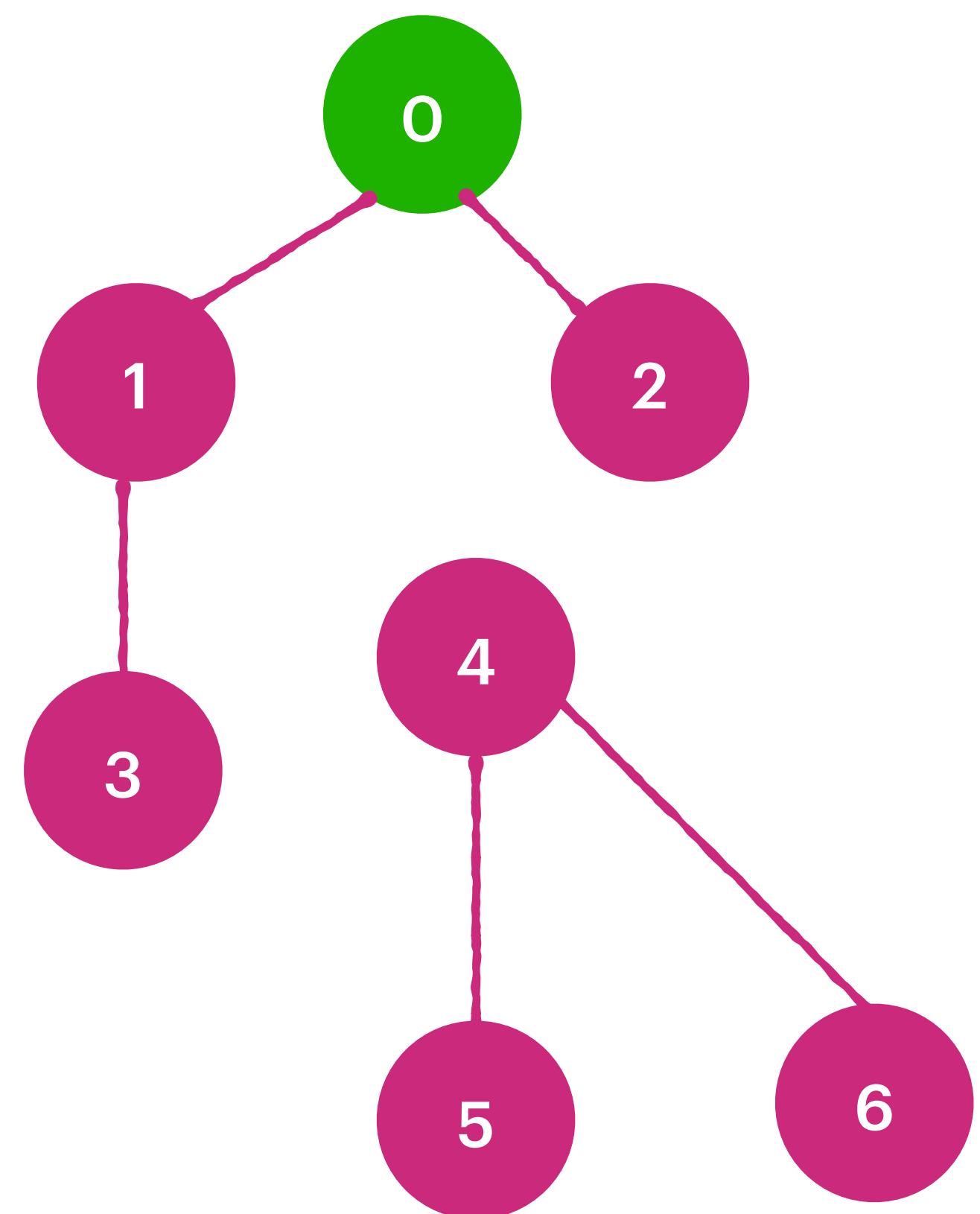
Is (0,4) connected ?  
 $\text{root}(0) = 0$

$\text{find}(4)$   
 $\text{find}(3)$   
 $\text{find}(2)$   
 $\text{find}(1)$   
 $\text{find}(0) = 0$

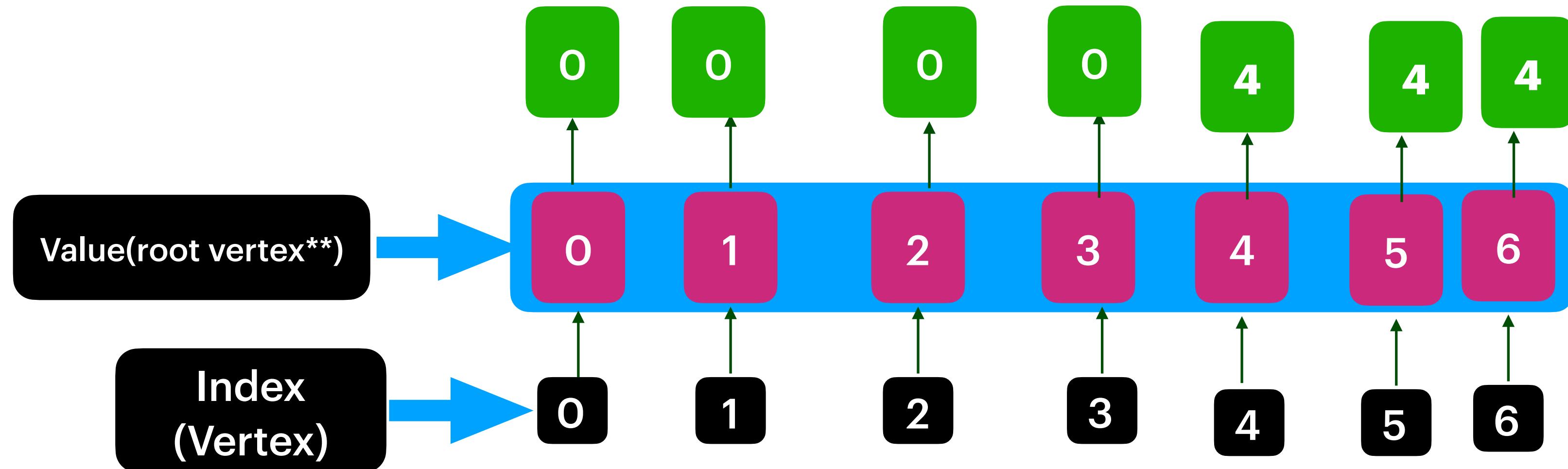
**find(int) Operation takes linear time If the Graph/Tree is skewed :**  
 Skewed means a root either has one child or no child.

## Quick Find :





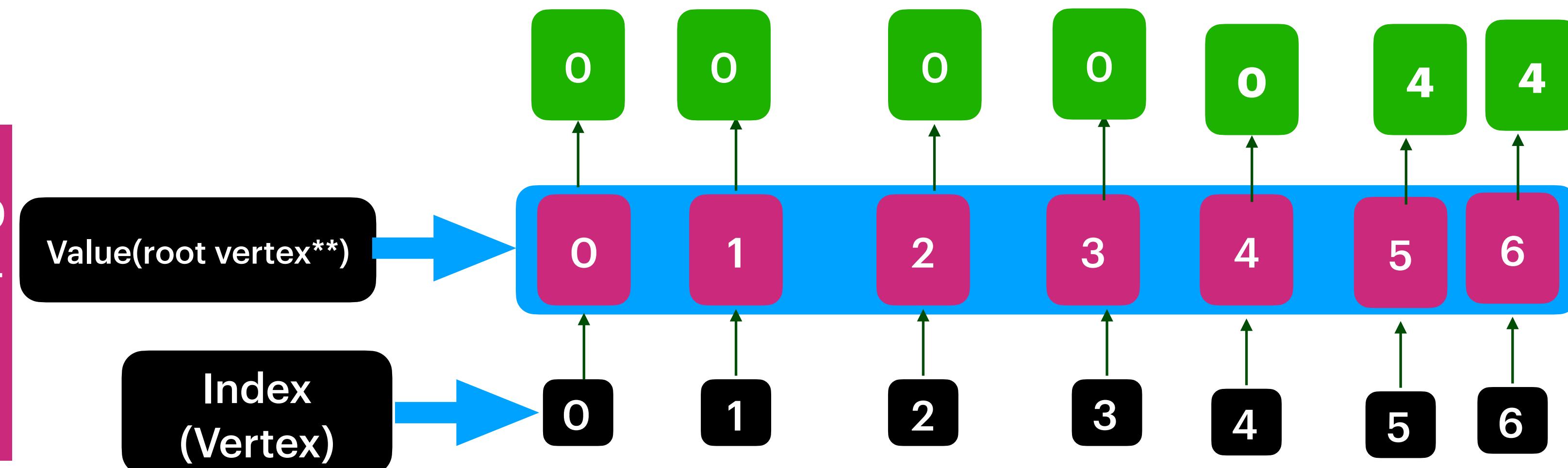
## Quick Union :

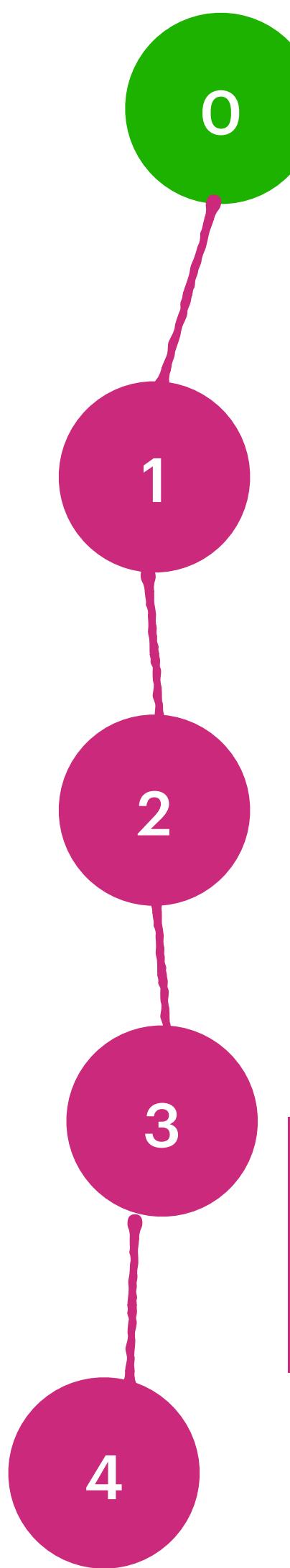


union(0,1)  
union(0,2)  
union(1,3)  
union(4,5)  
union(4,6)

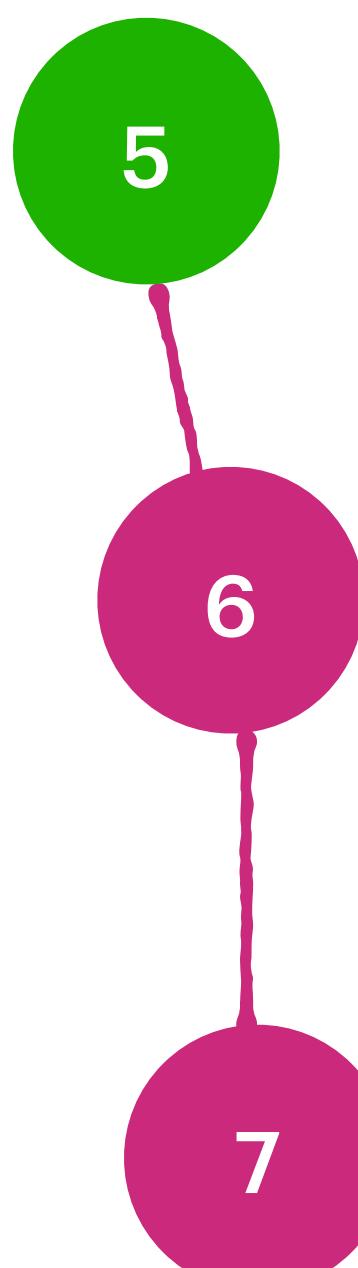
union(0, 1)  
Int rootX = find(x); // 0  
Int rootY = find(y); // 1  
root[rootY] = rootX; //

After union(3,5)  
Int rootX = find(3) = 0  
Int rootY = find(5) = 4  
root[rootY] = rootX  
root[4] = 0

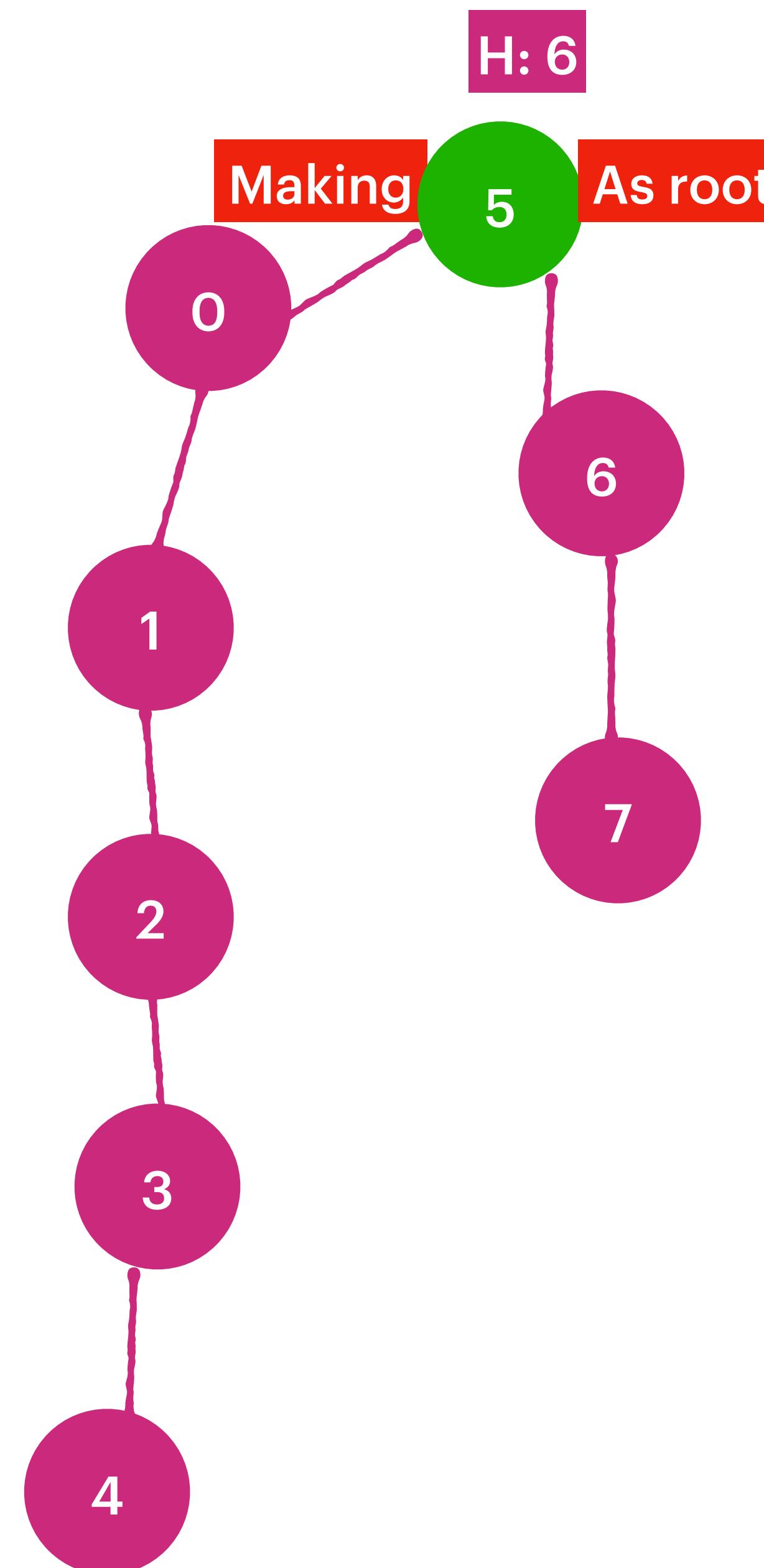




H: 5

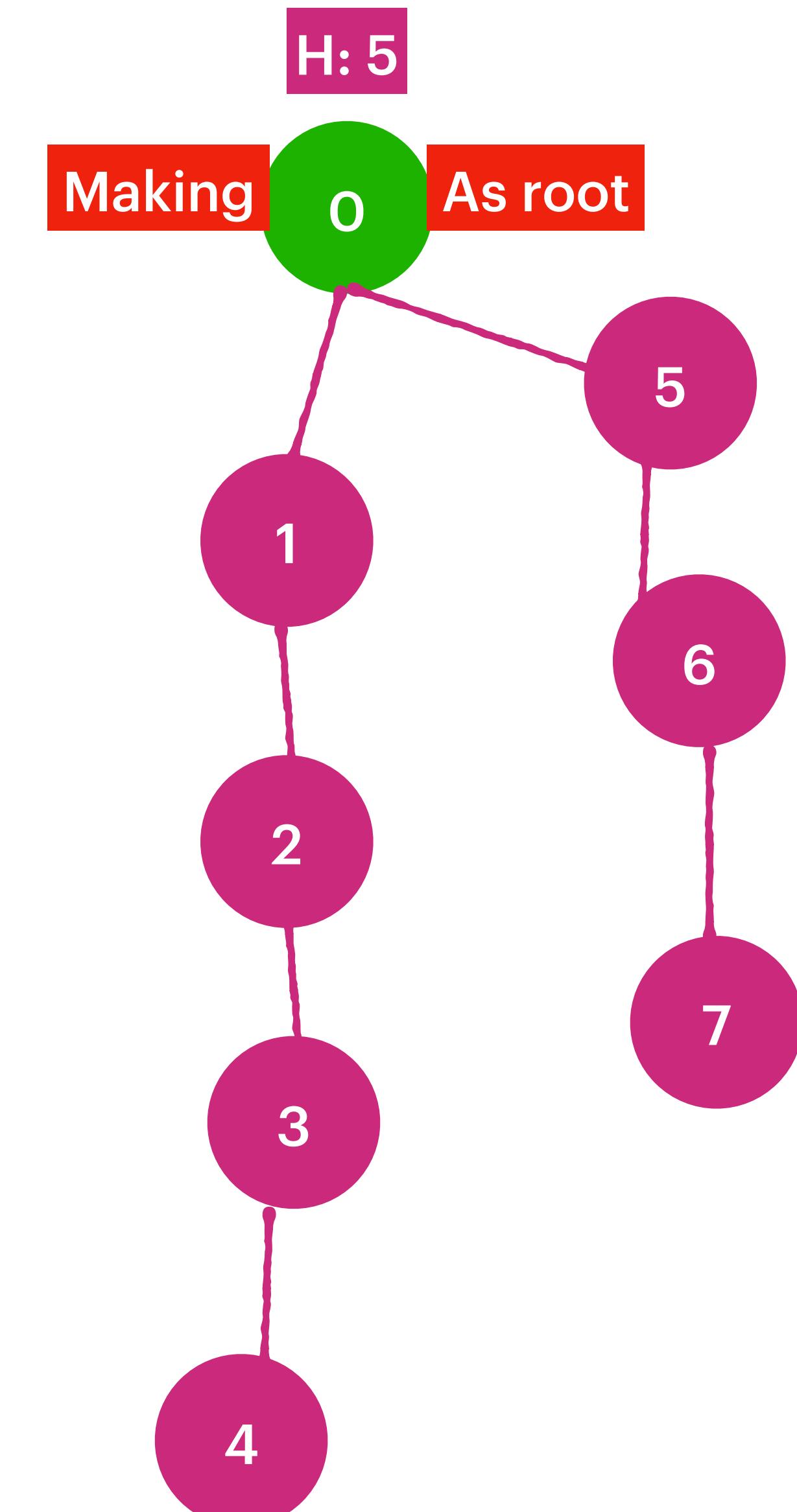


H: 3



Making

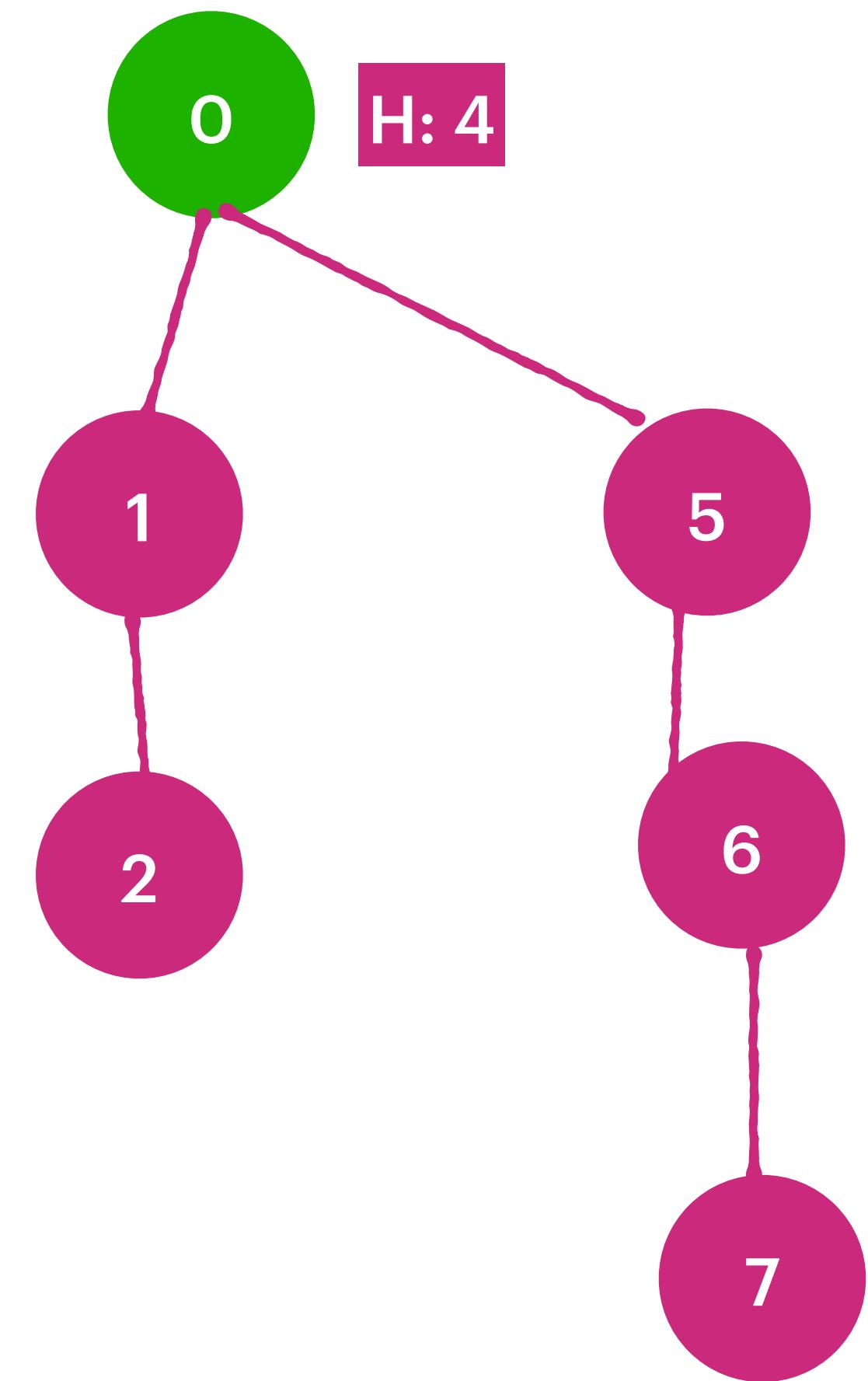
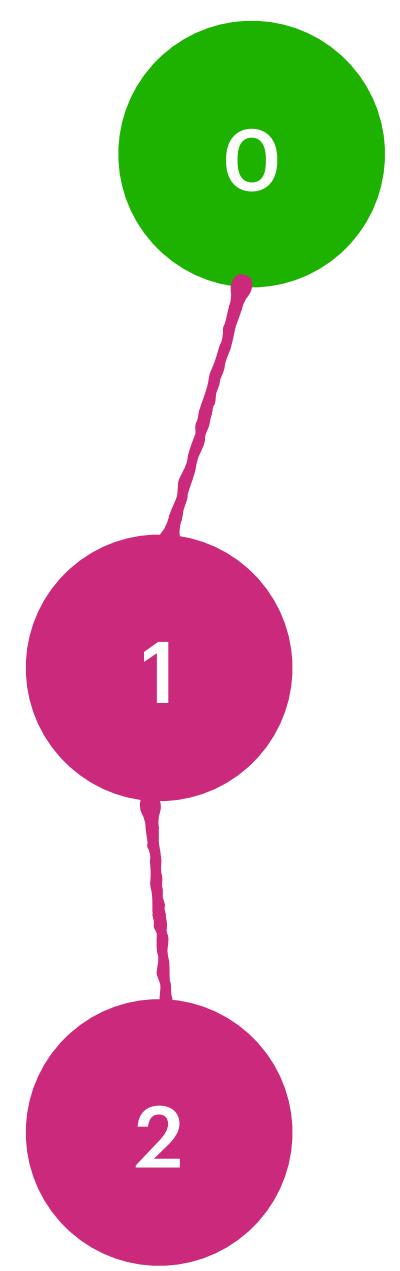
As root

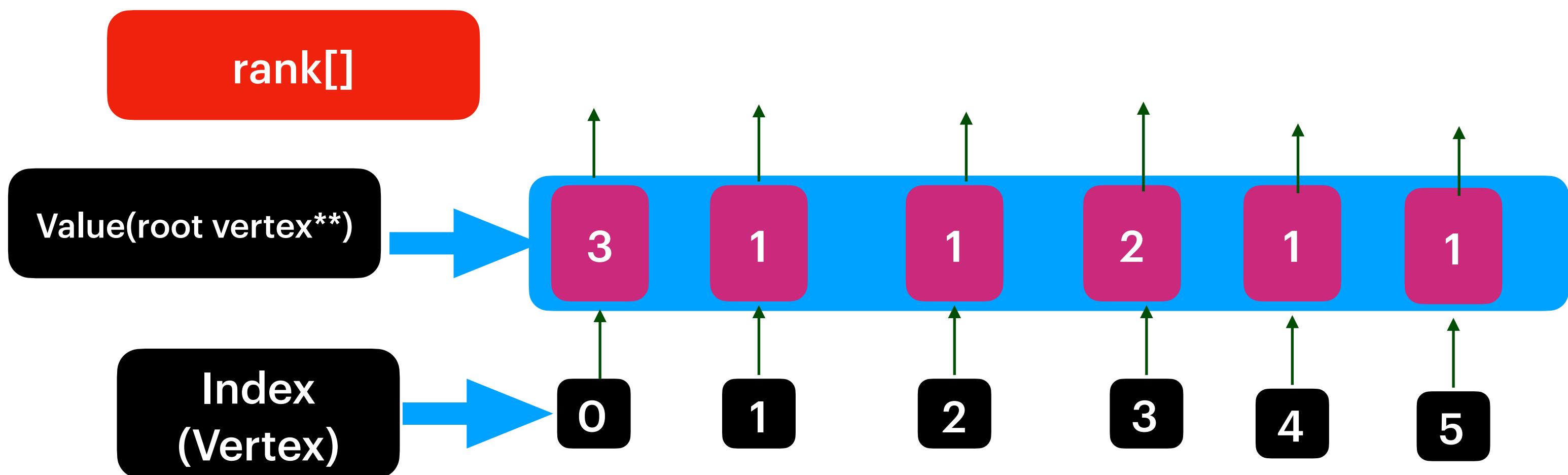
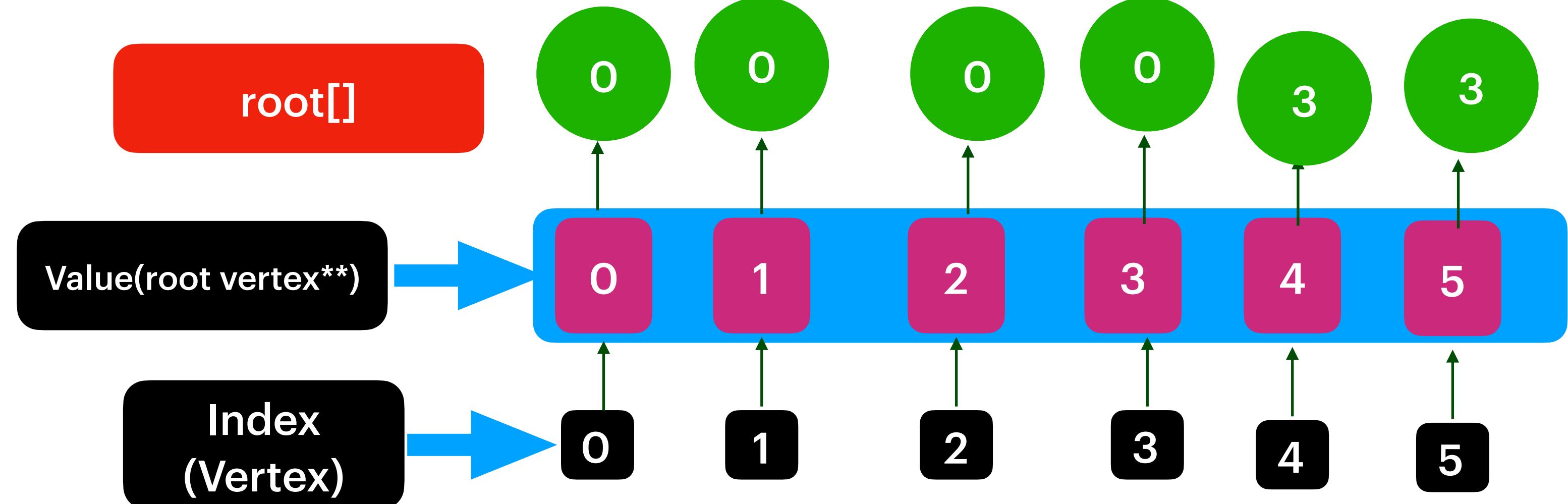
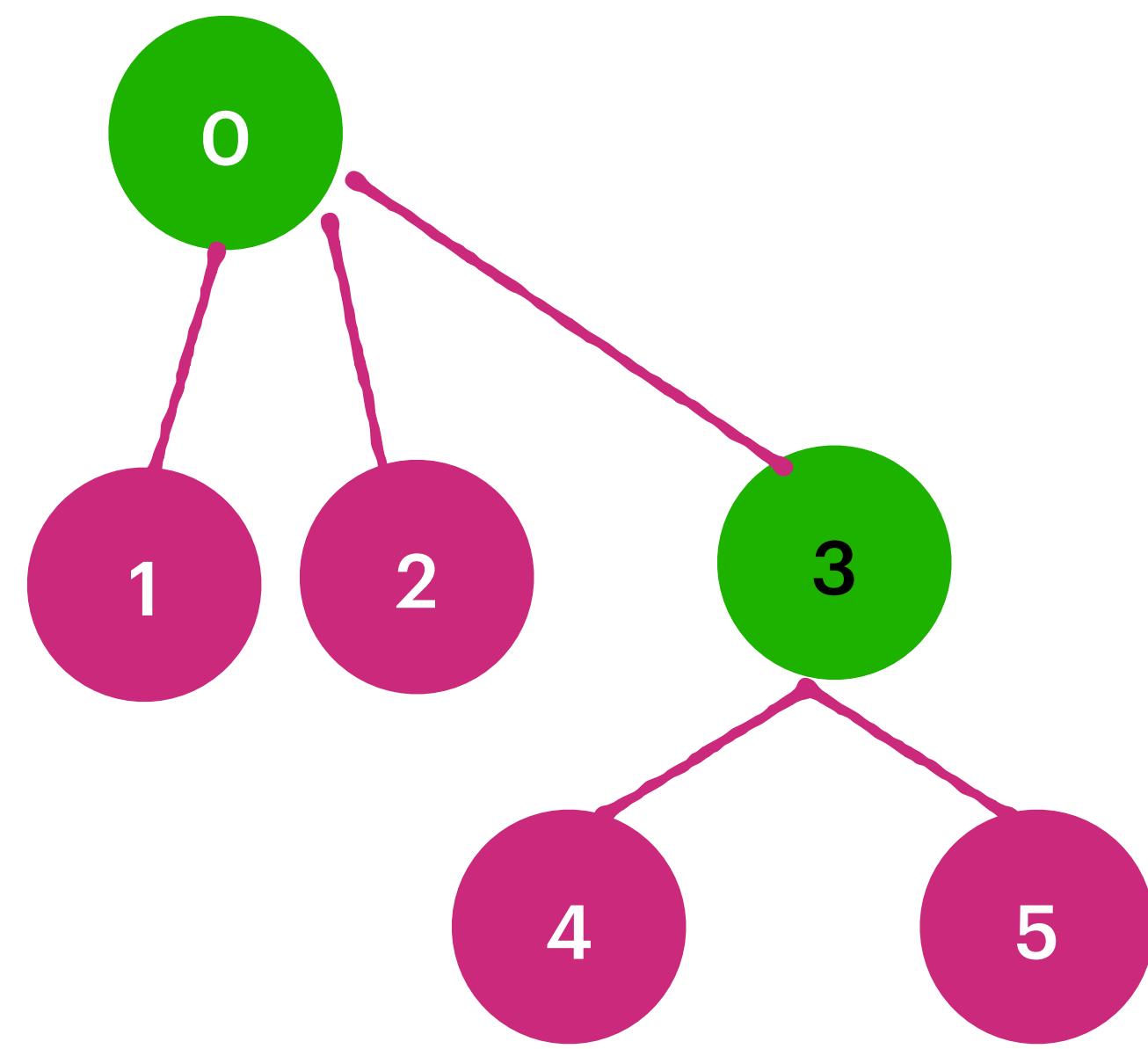


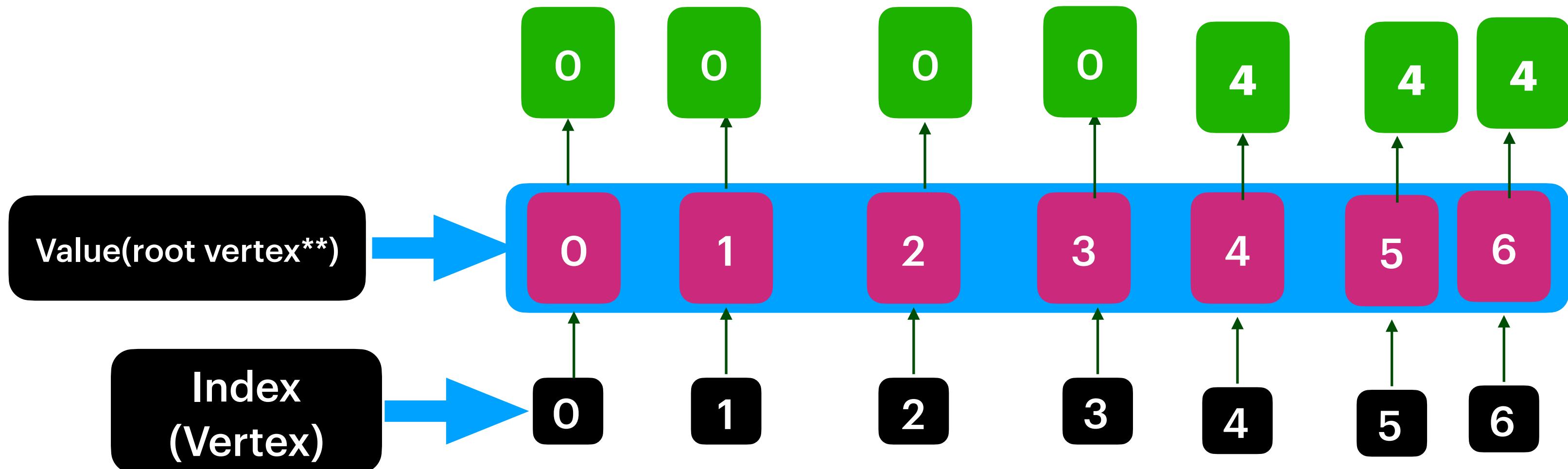
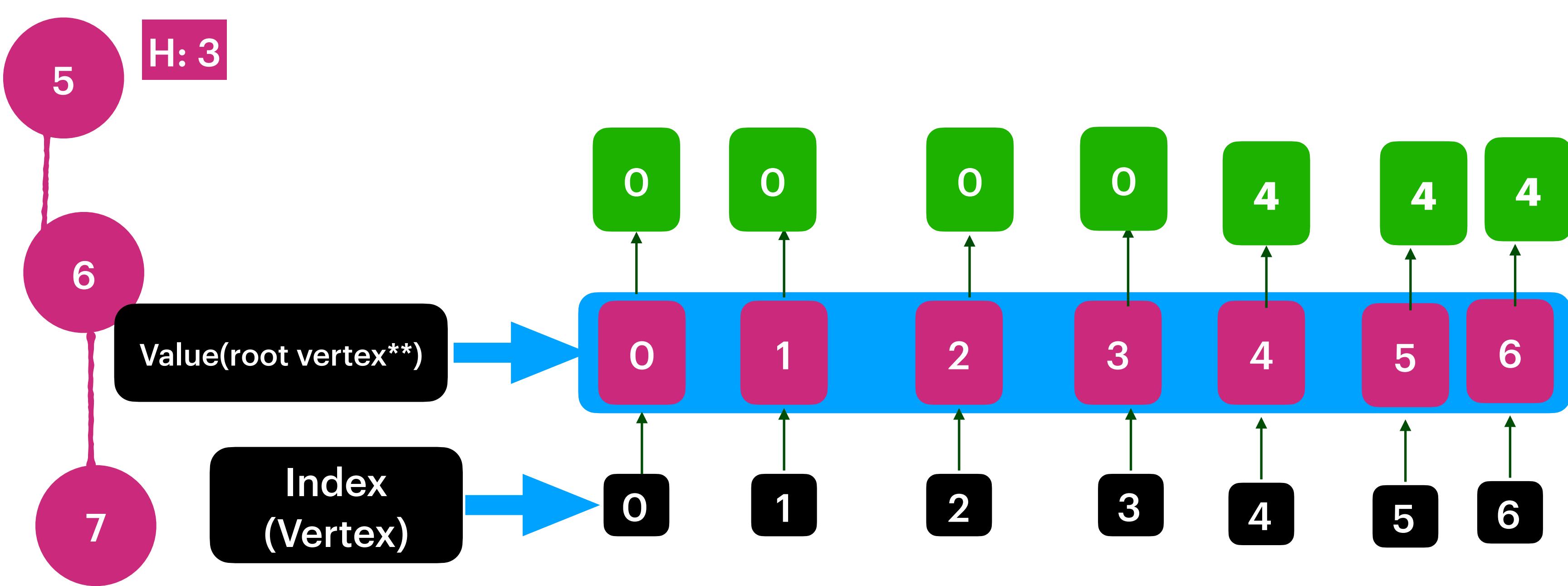
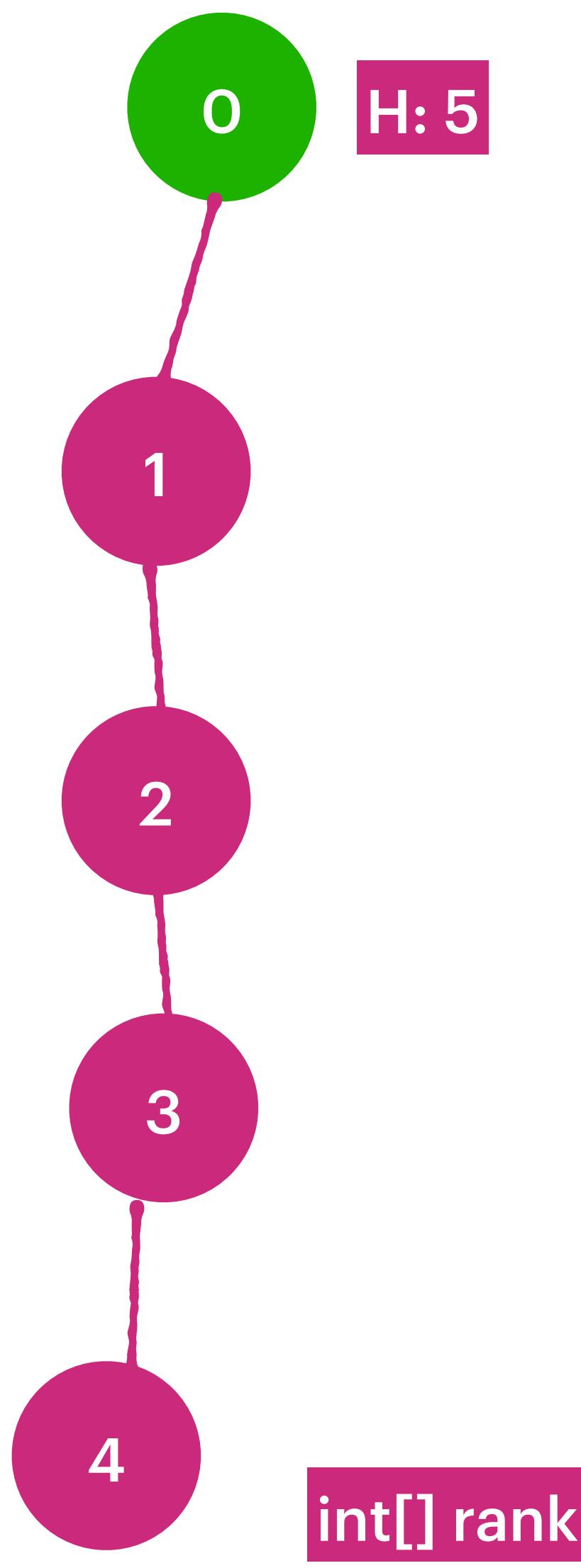
Making

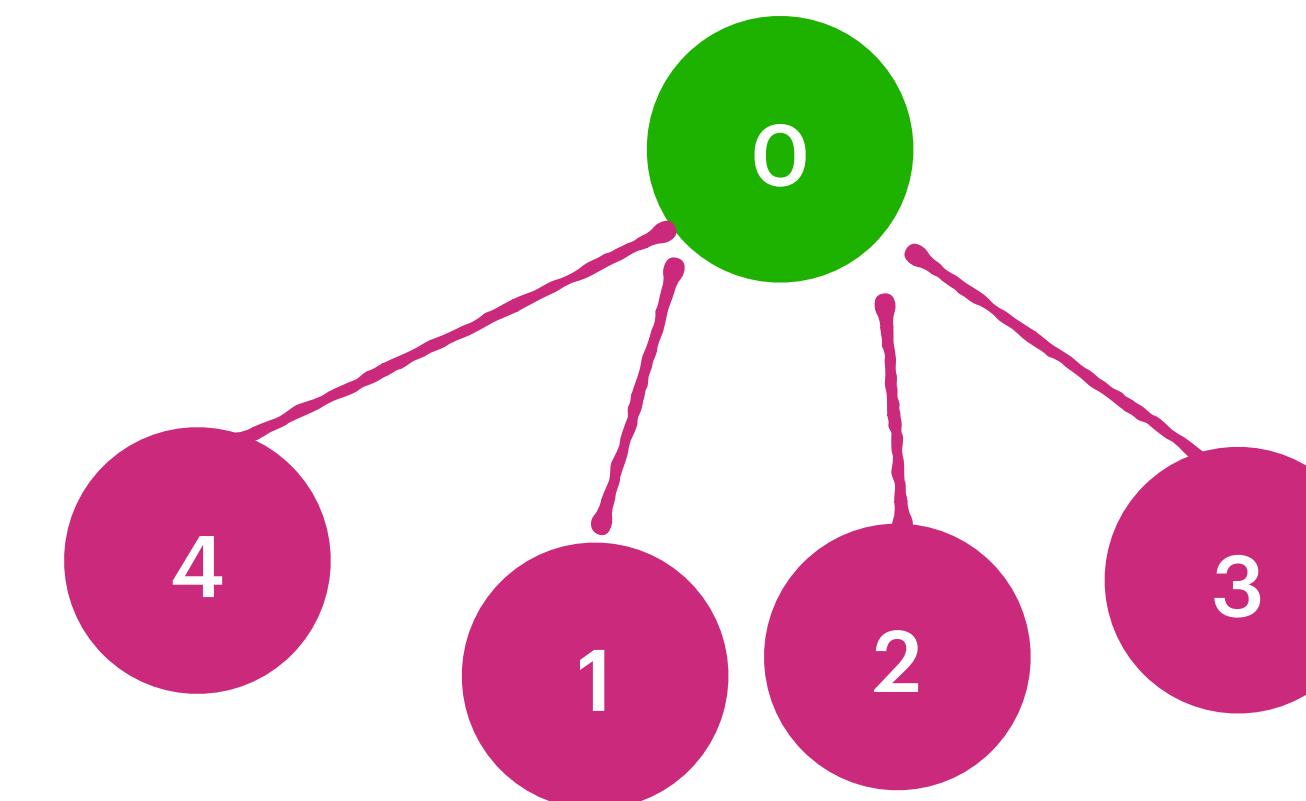
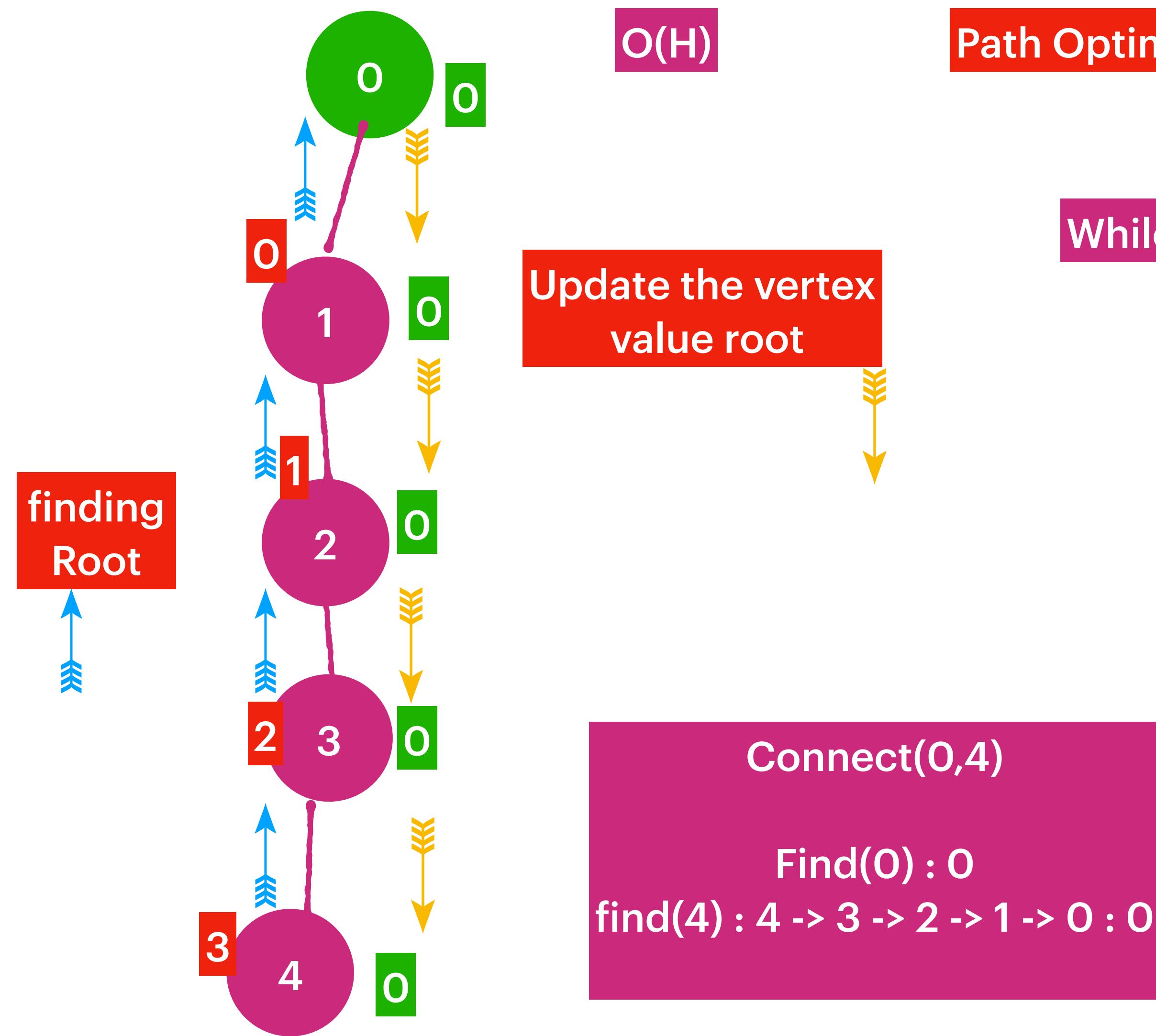
As root

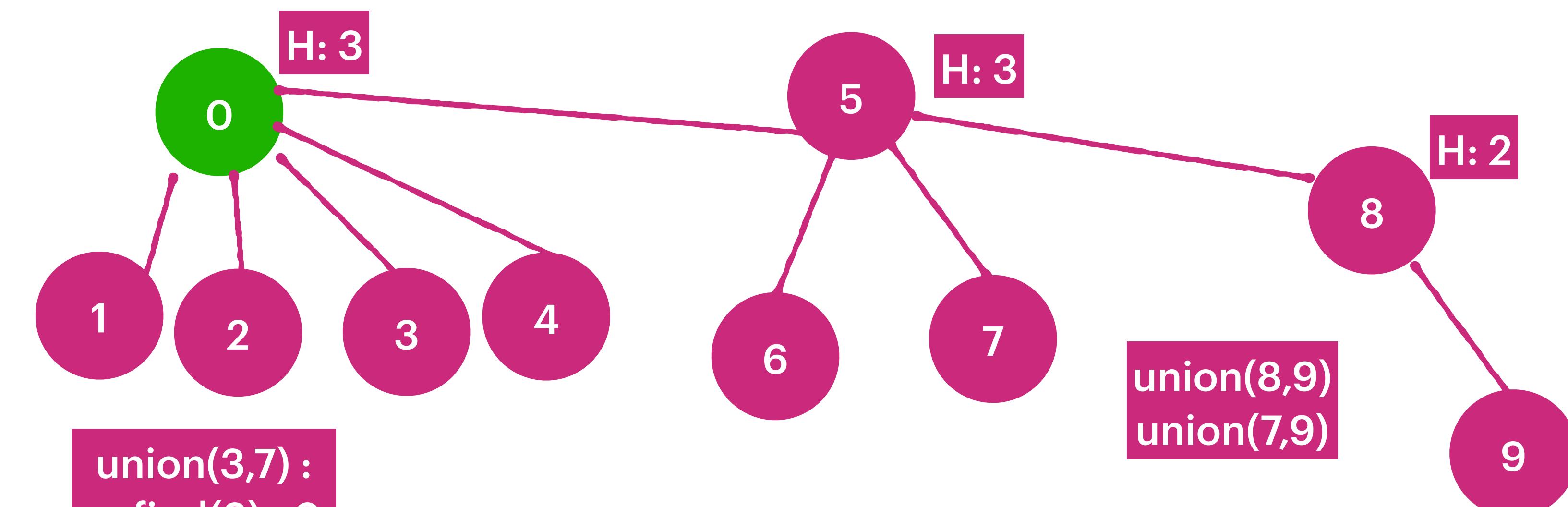
**Ranking :**  
While Joining always assign  
Mak Height Root node as Parent.



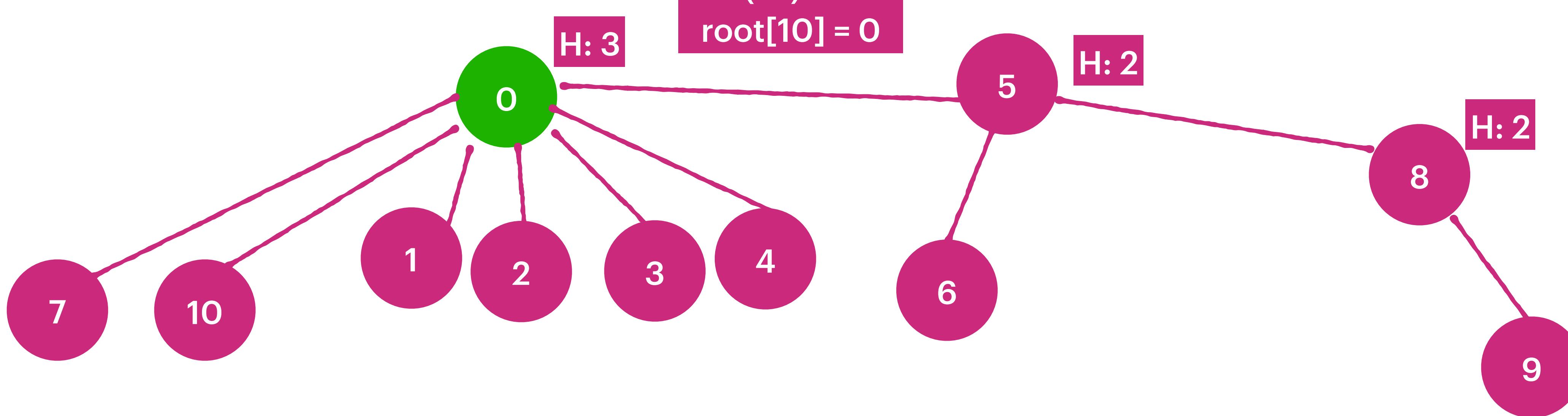


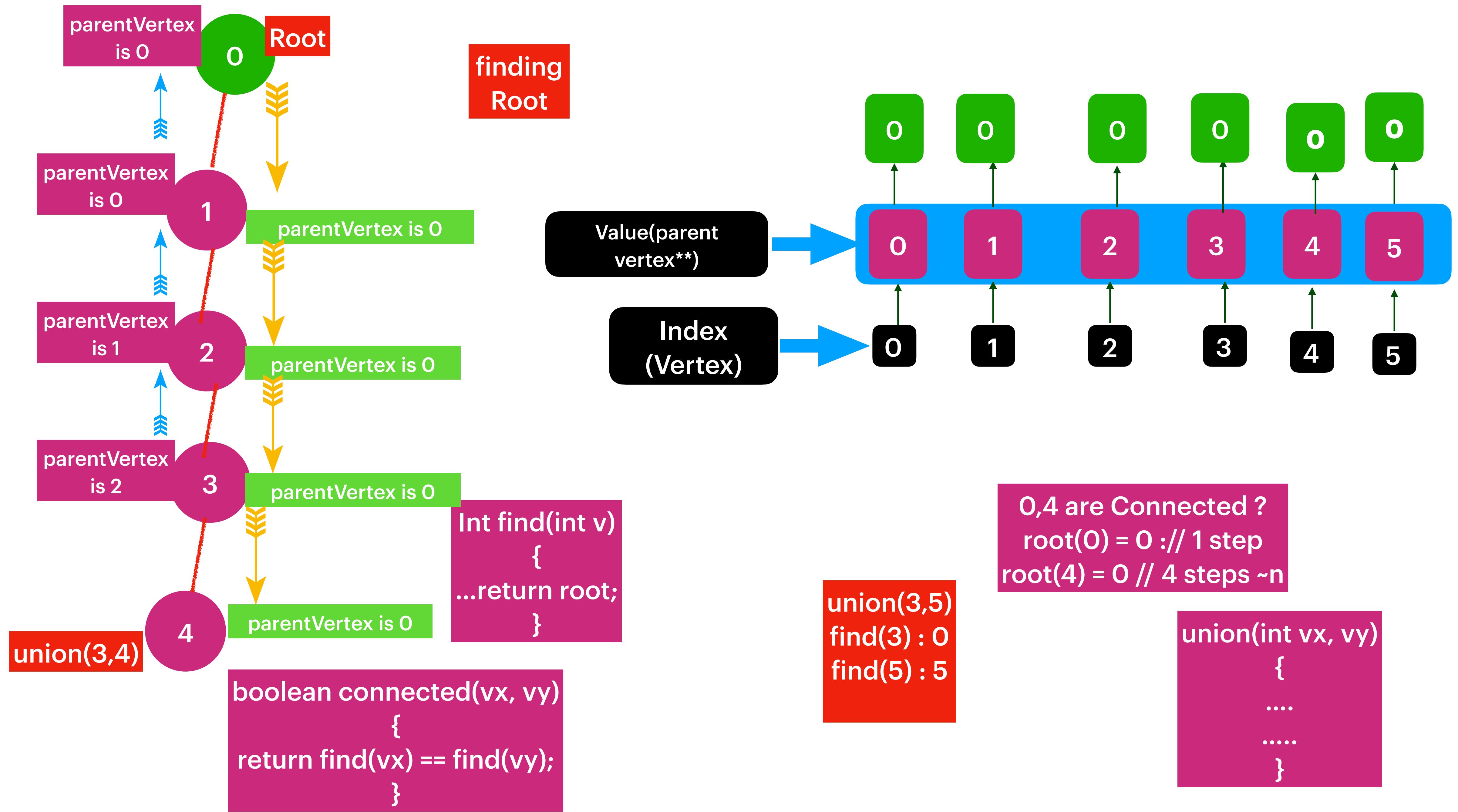


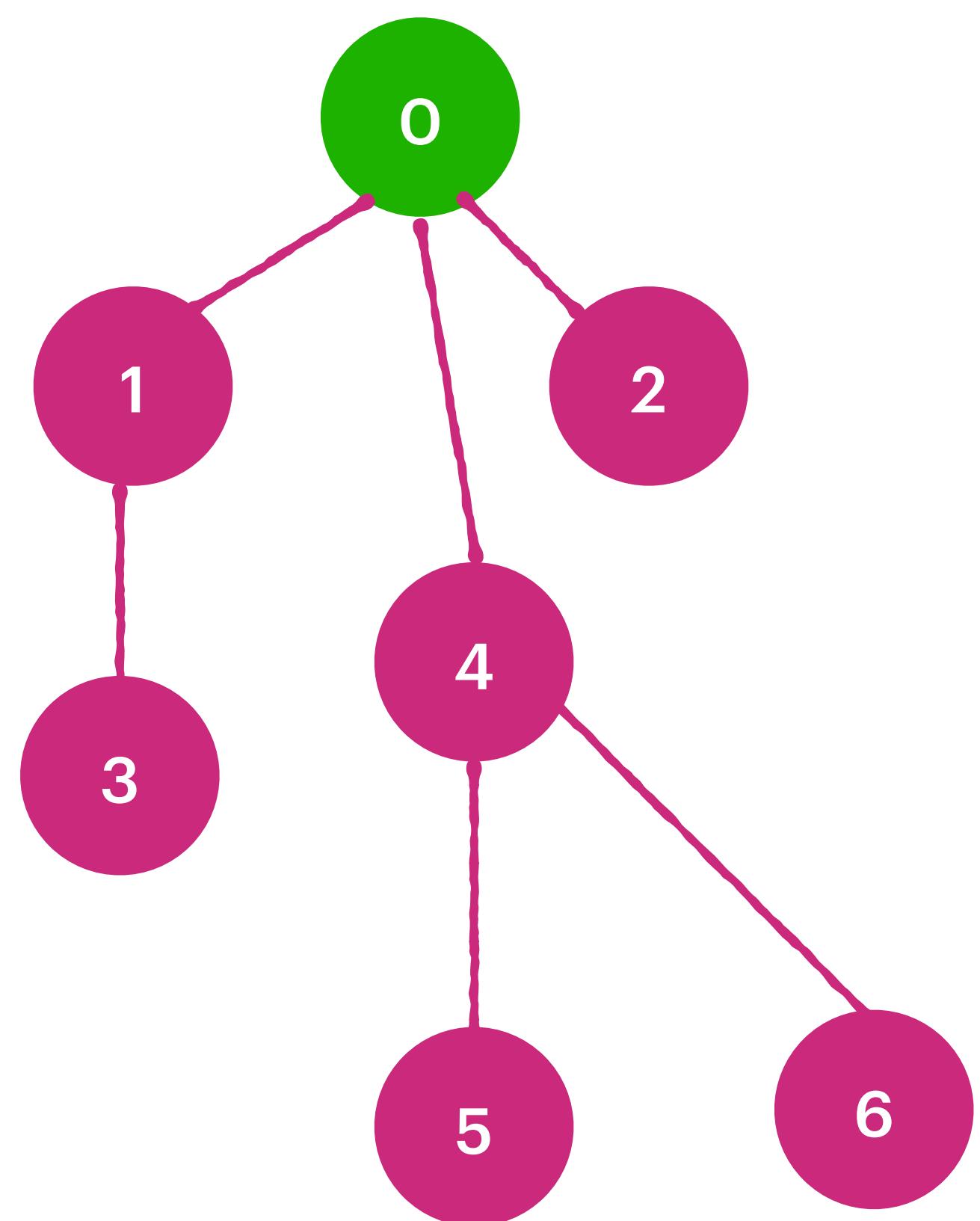




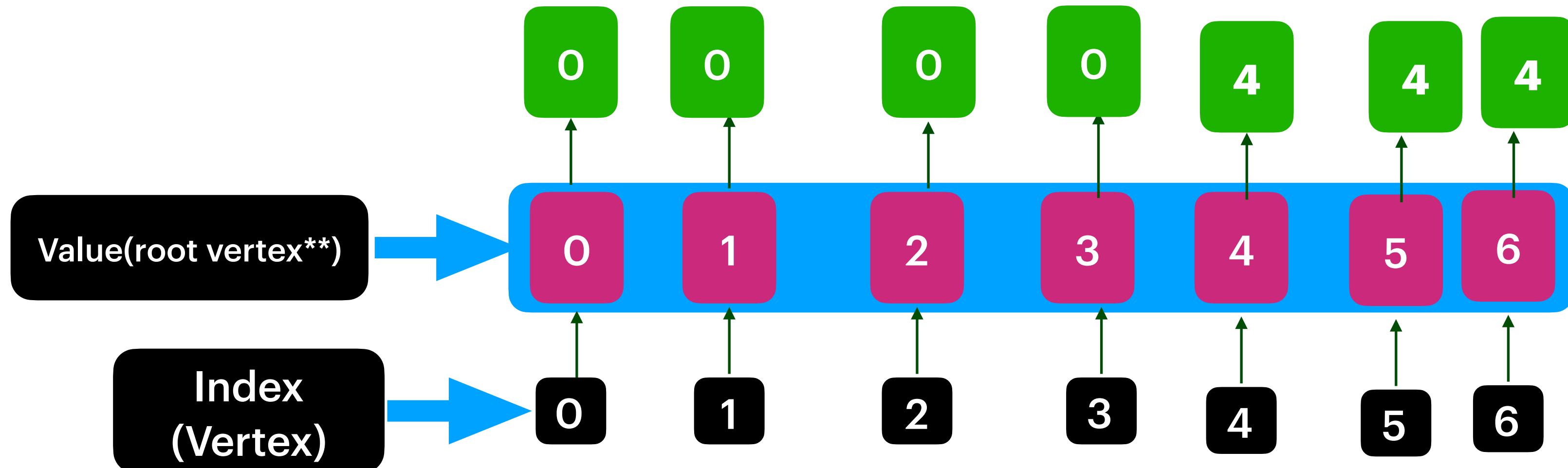
**isConnected(1,9) :**  
 $\text{find}(1) : 0 // 1\text{step}$   
 $\text{find}(9) : 8 \rightarrow 5 \rightarrow 0 : 0 // 3\text{steps}$   
 returns true.







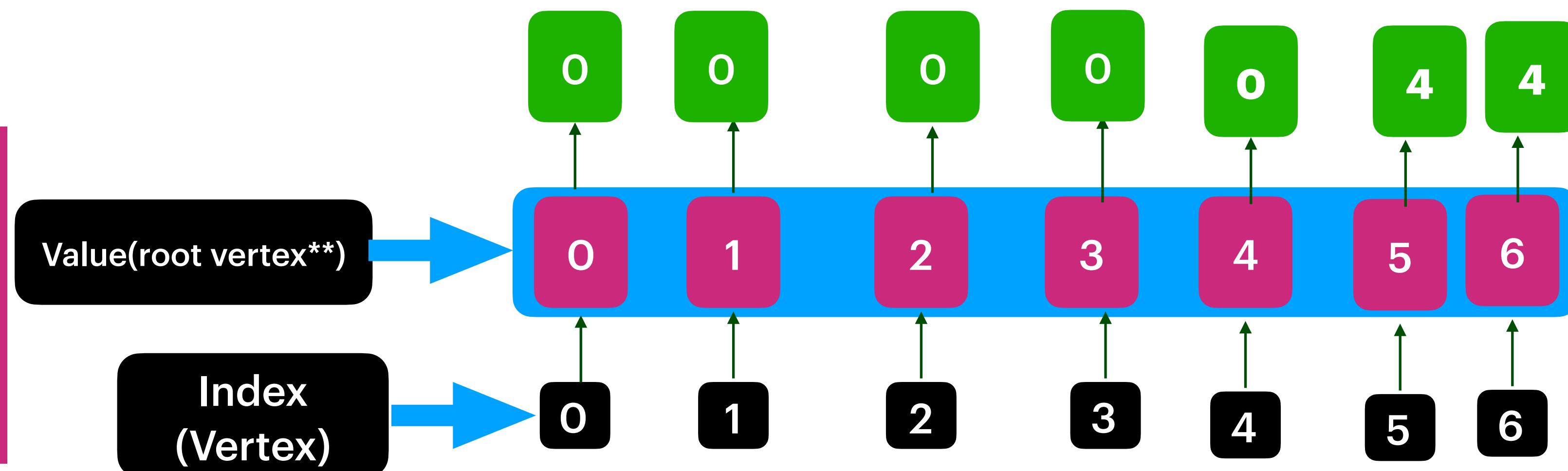
## Quick Union :

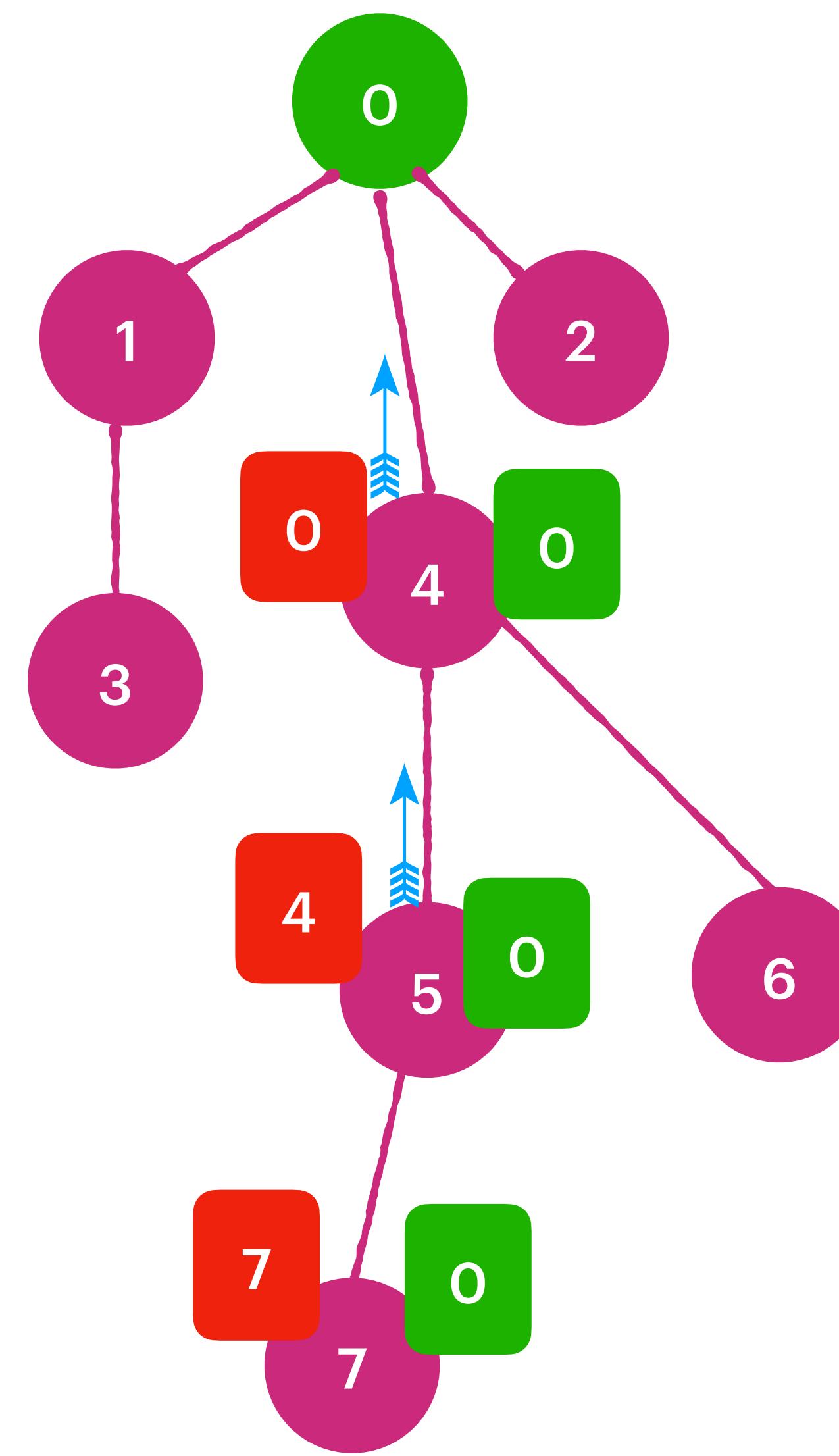


union(0,1)  
union(0,2)  
union(1,3)  
union(4,5)  
union(4,6)

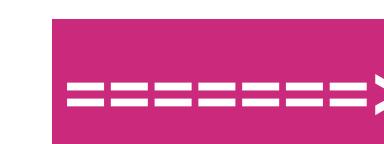
union(0, 1)  
Int rootX = find(x); // 0  
Int rootY = find(y); // 1  
root[rootY] = rootX; //

After union(3,5)  
Int rootX = find(3) = 0  
Int rootY = find(5) = 4  
root[rootY] = rootX  
root[4] = 0

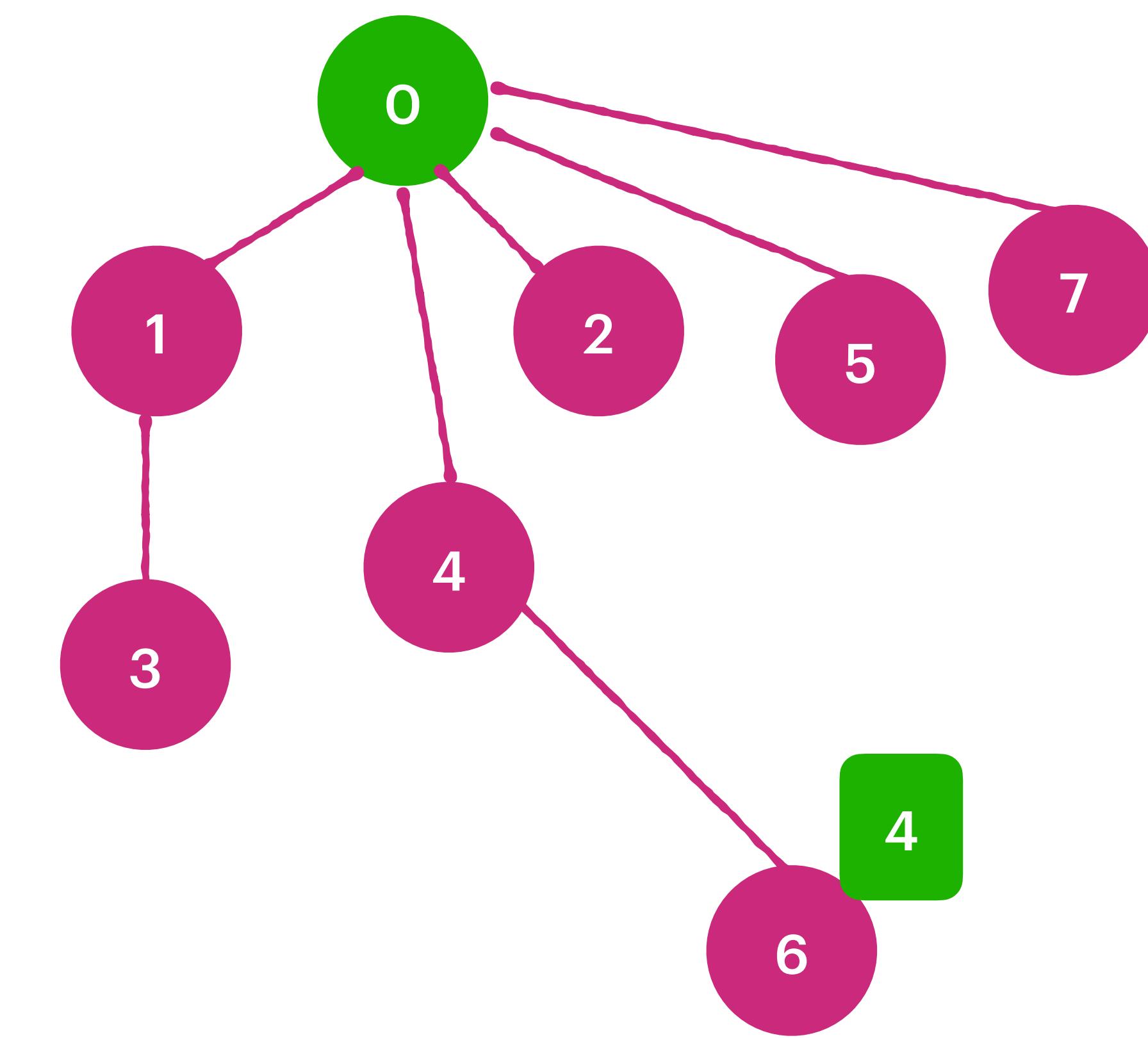




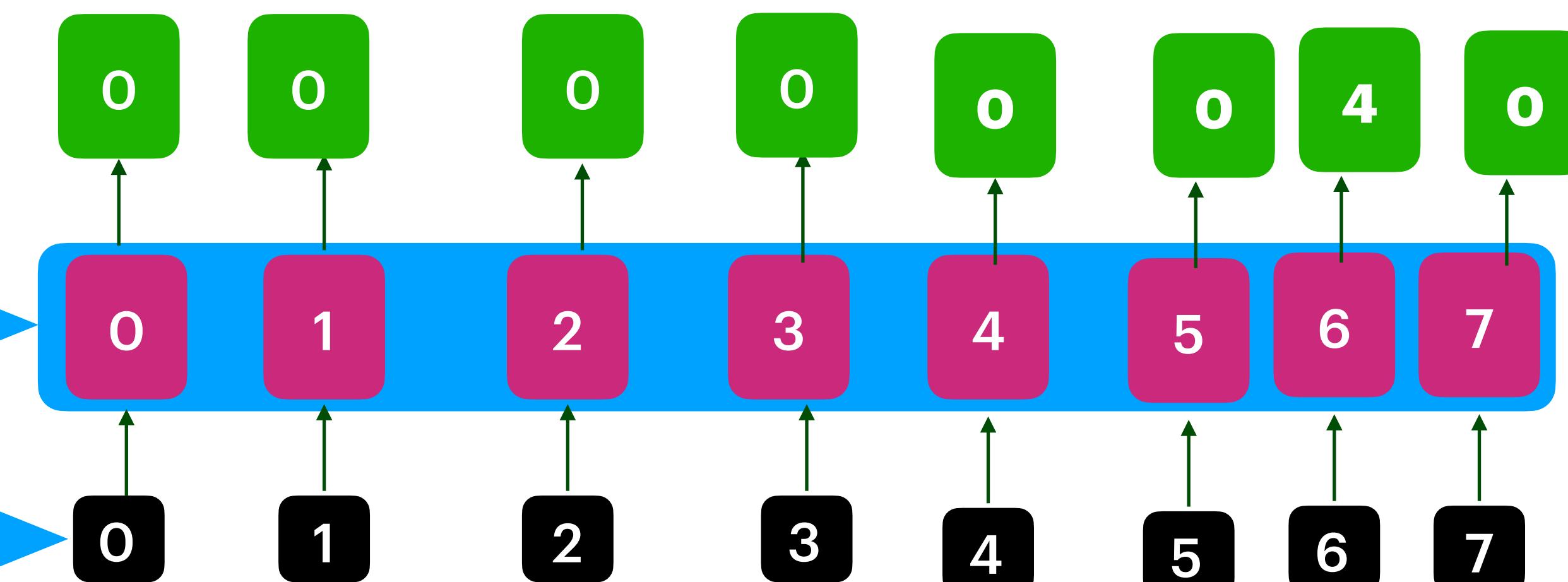
Quick Union :



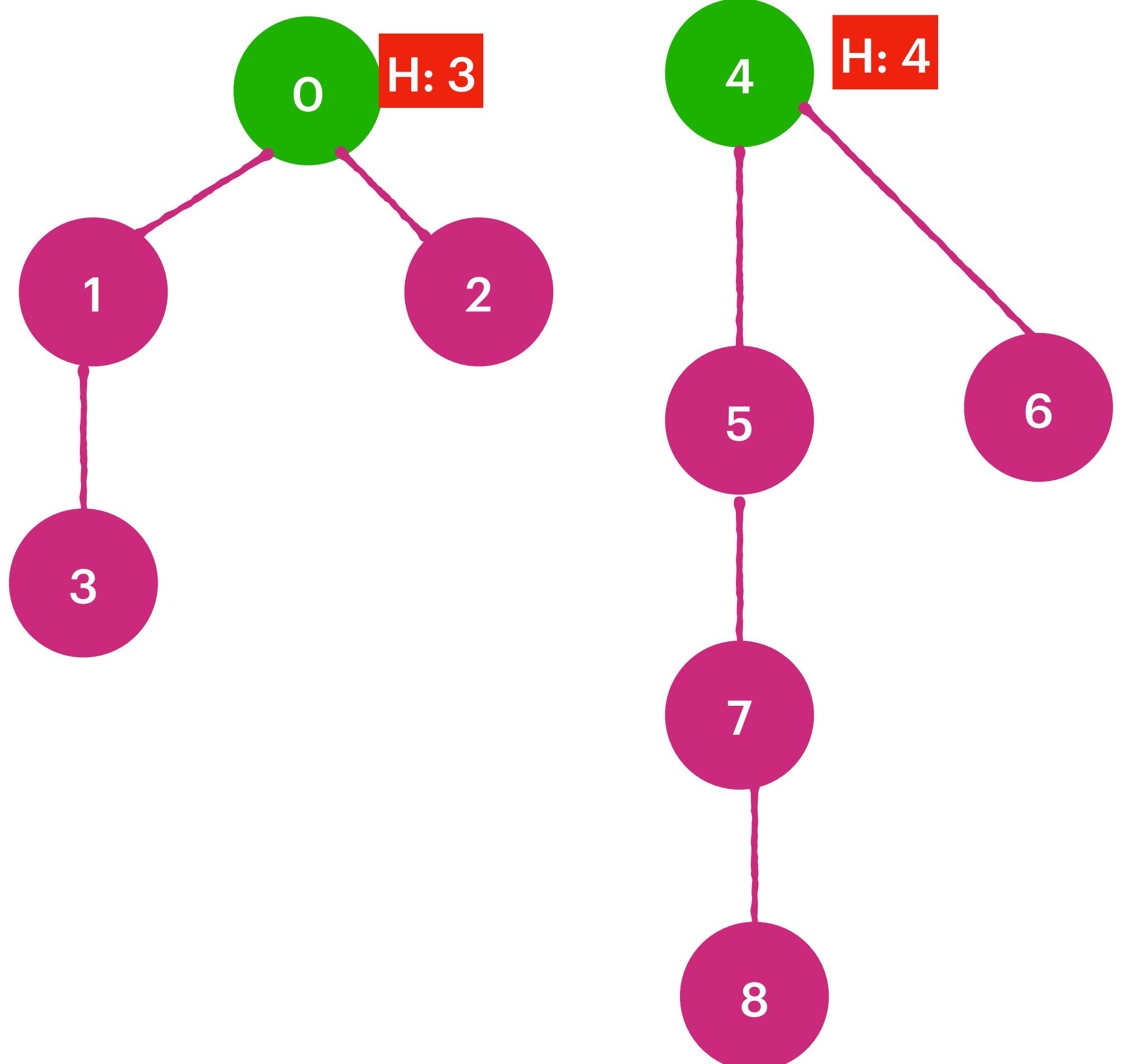
`union(5,7)`



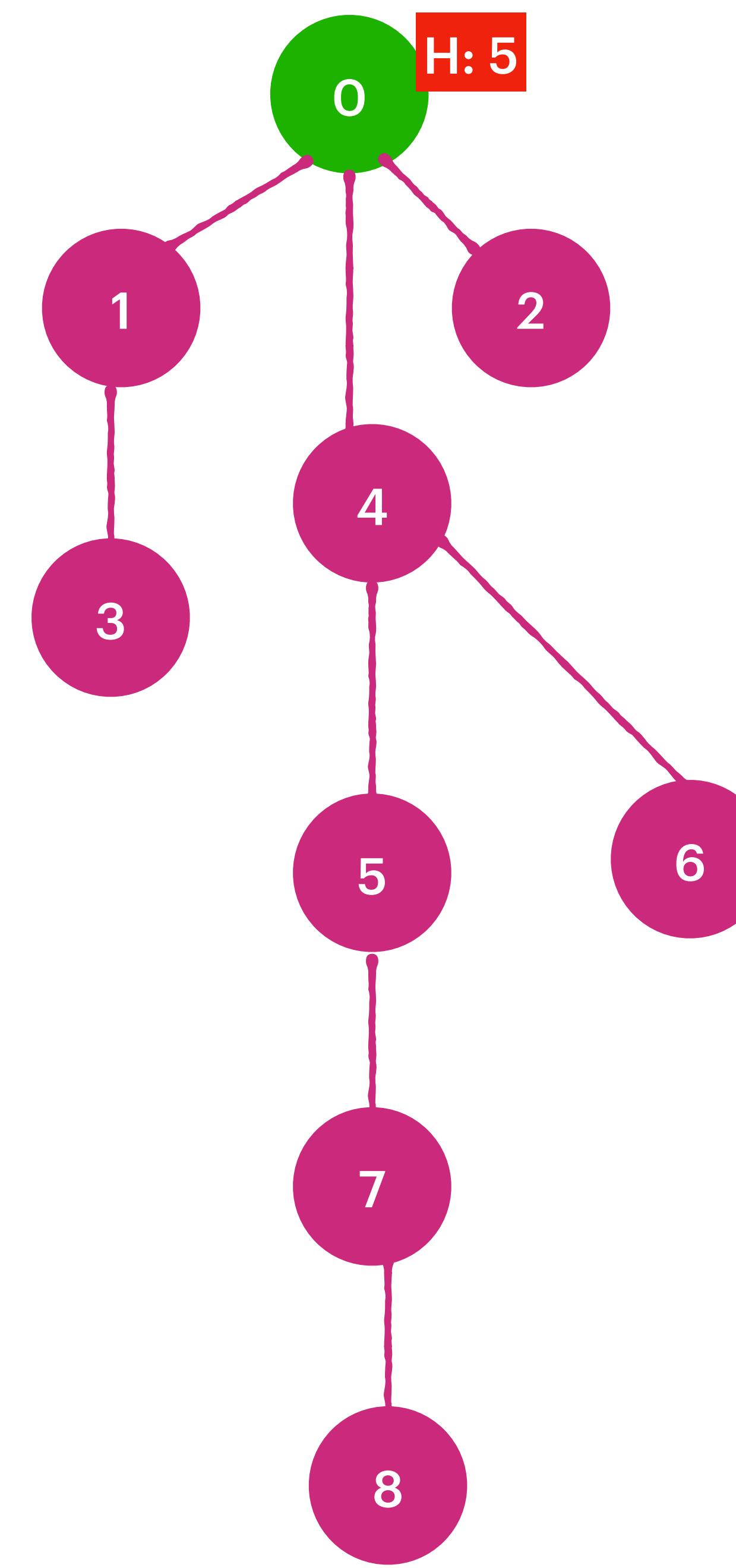
`Value(root vertex**)`

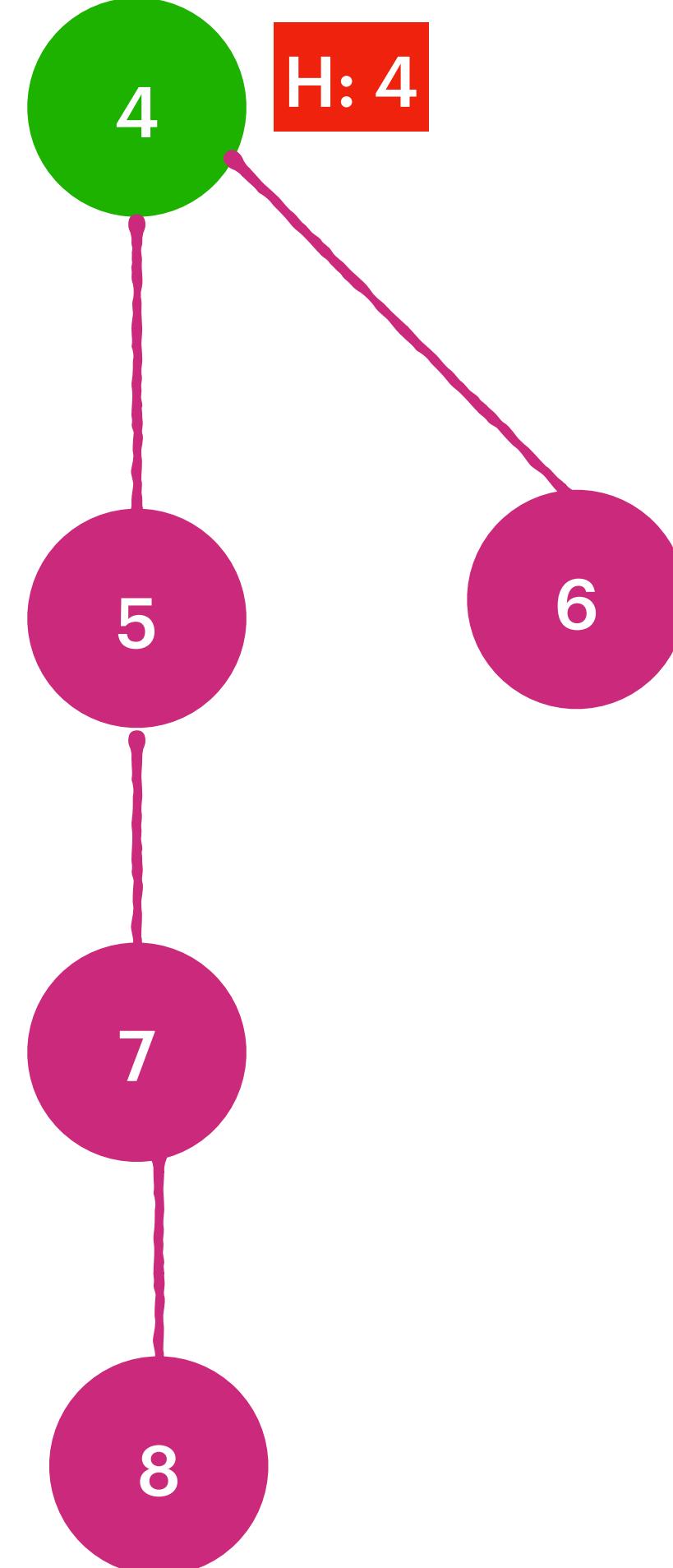
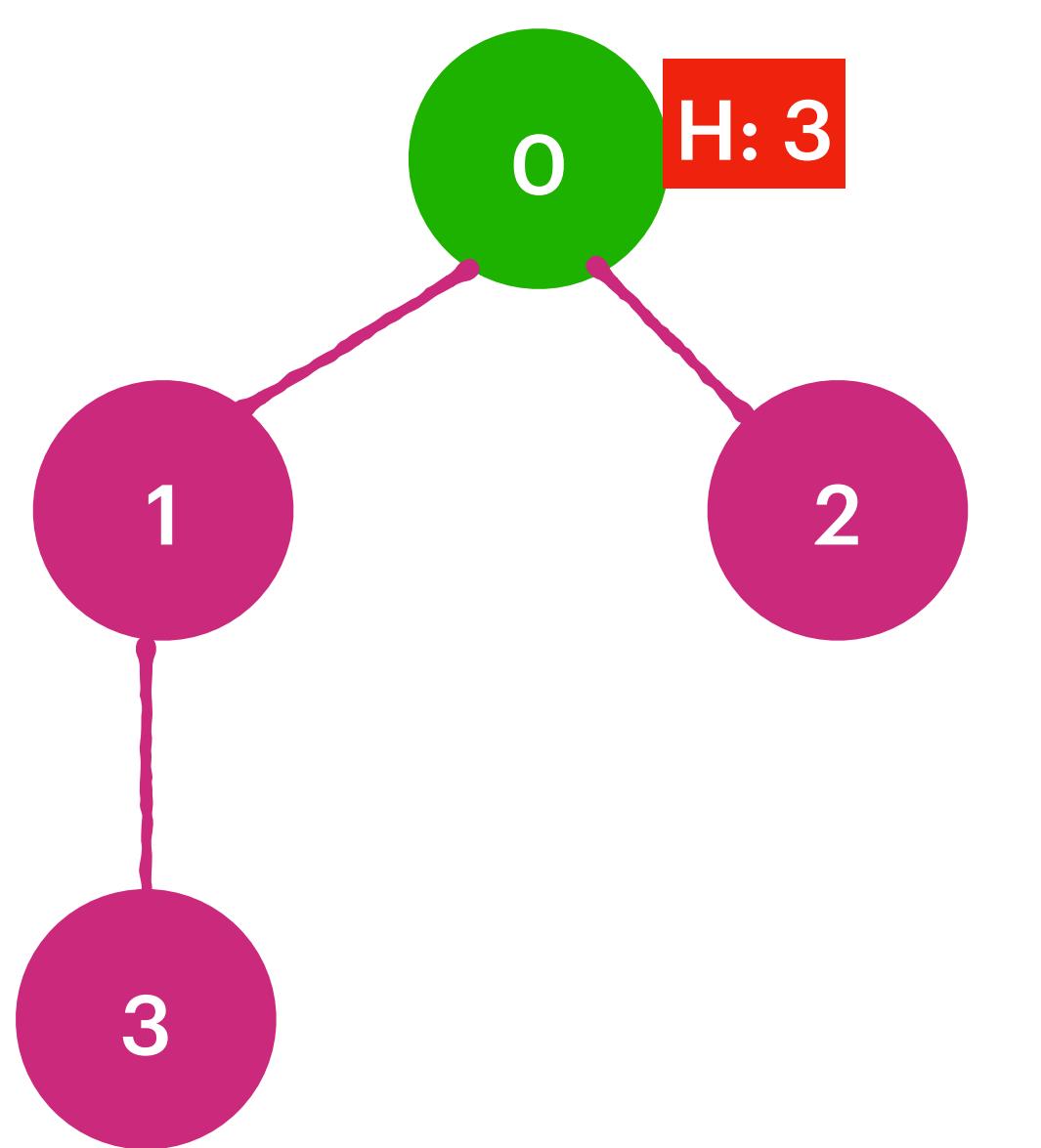


`Index  
(Vertex)`



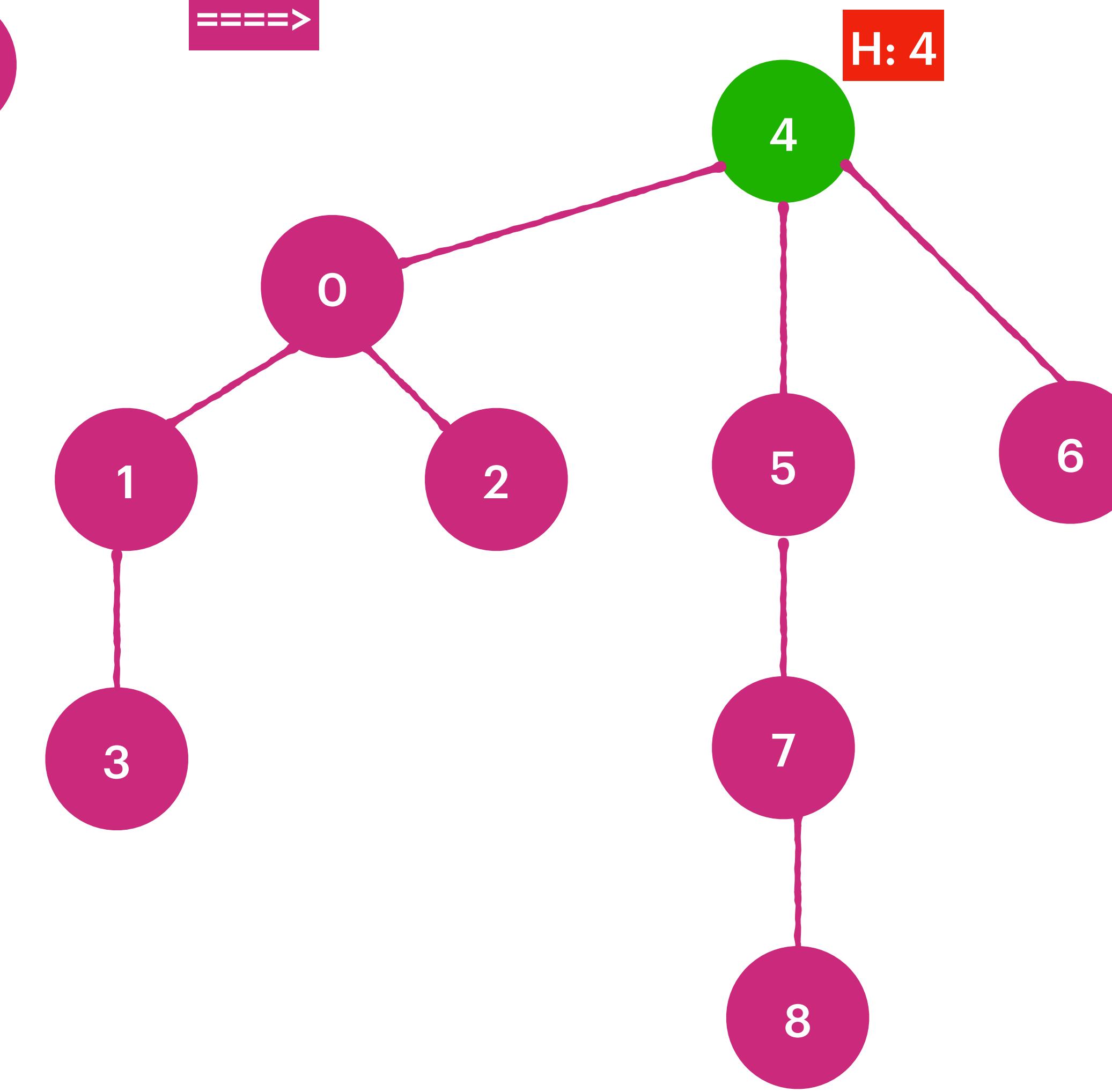
union(3,8)  
find(3) = 0  
find(8) = 4  
vetexs[4] = 0  
=====>

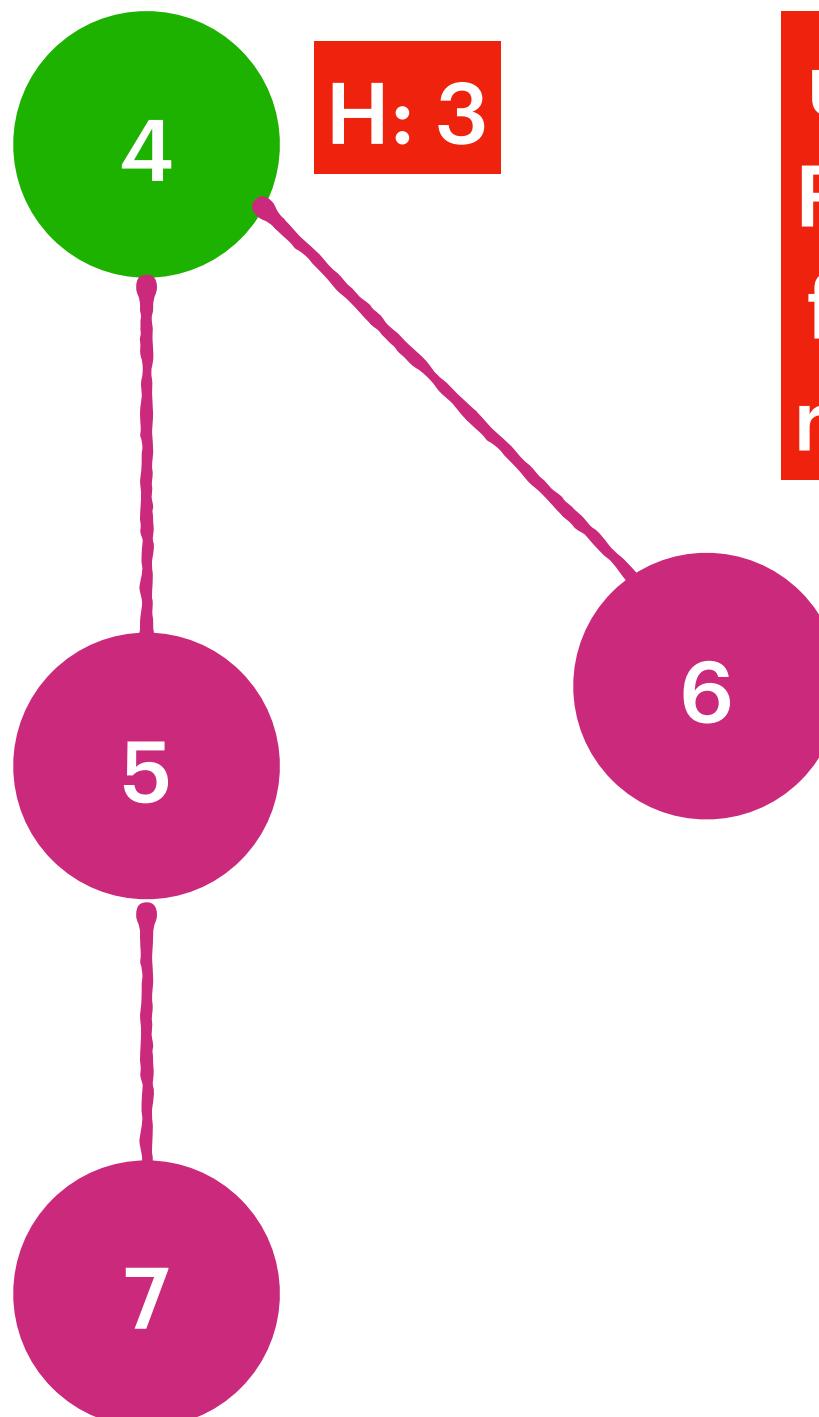
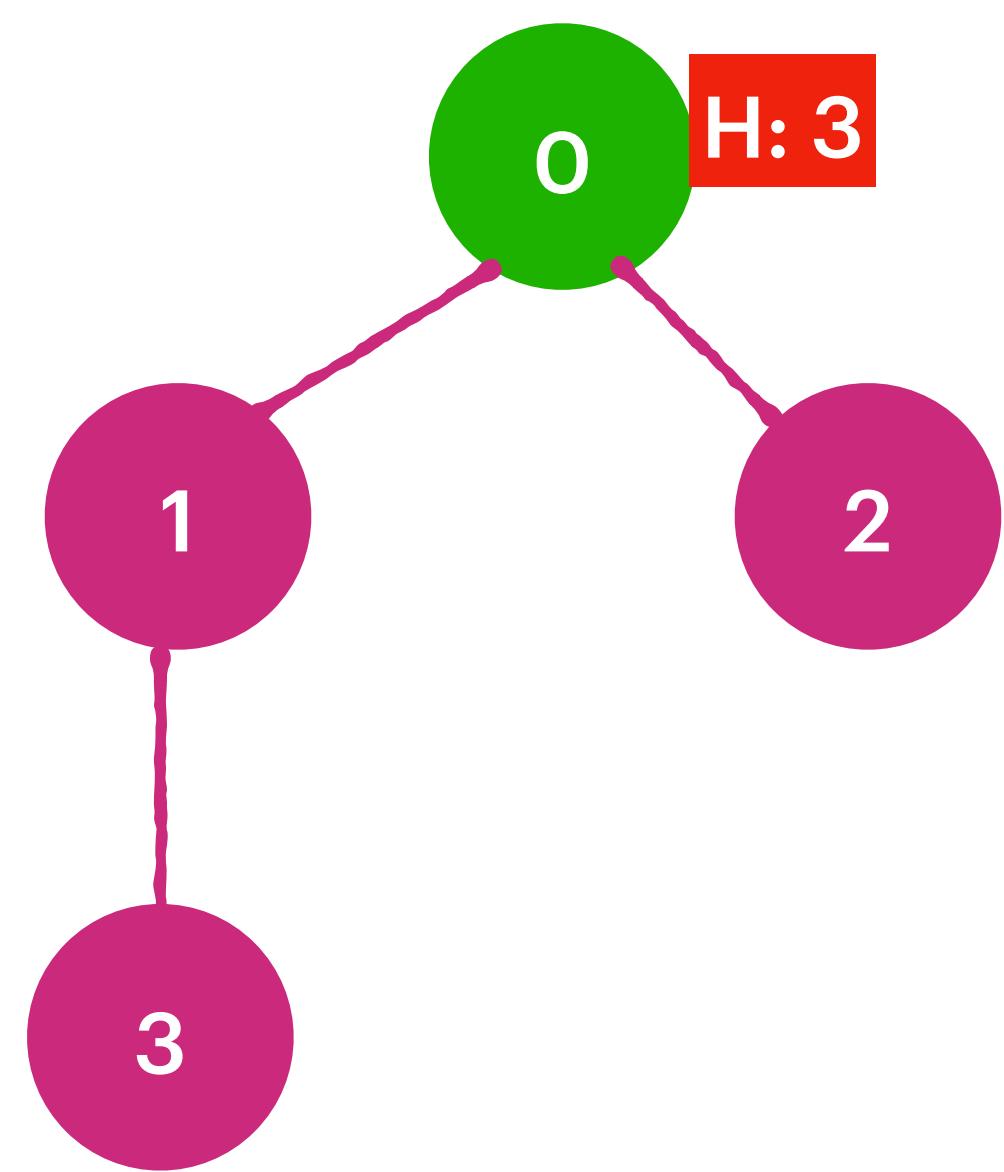




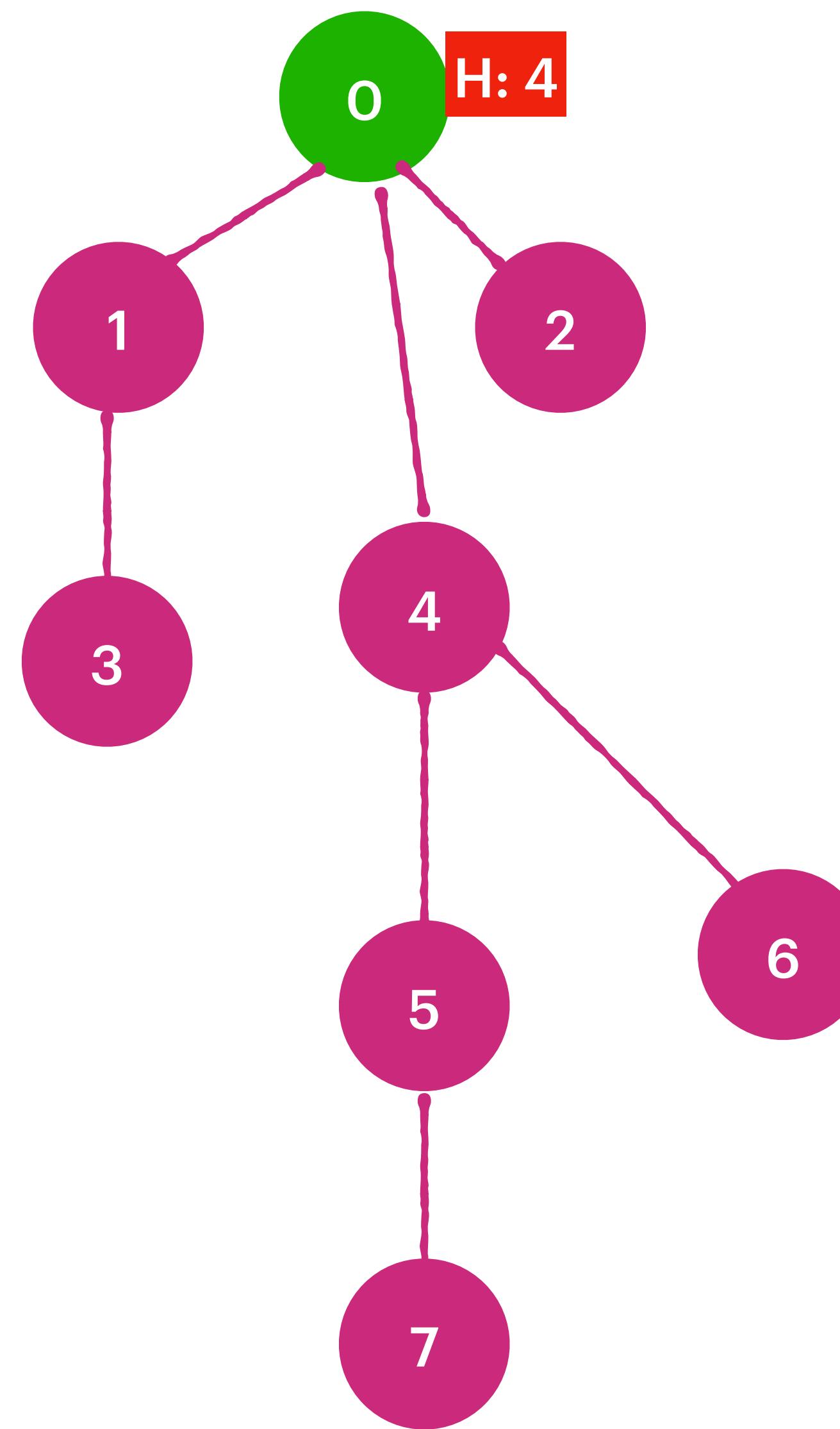
`union(3,8)`  
`find(3) = 0`  
`find(8) = 4`  
`vetexs[0] = 4`

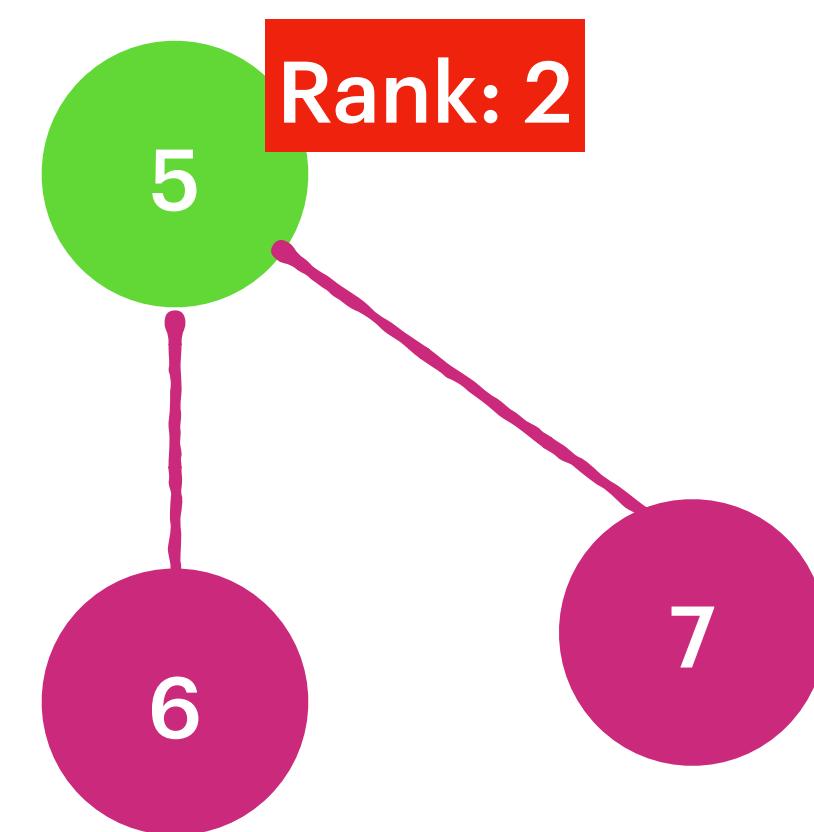
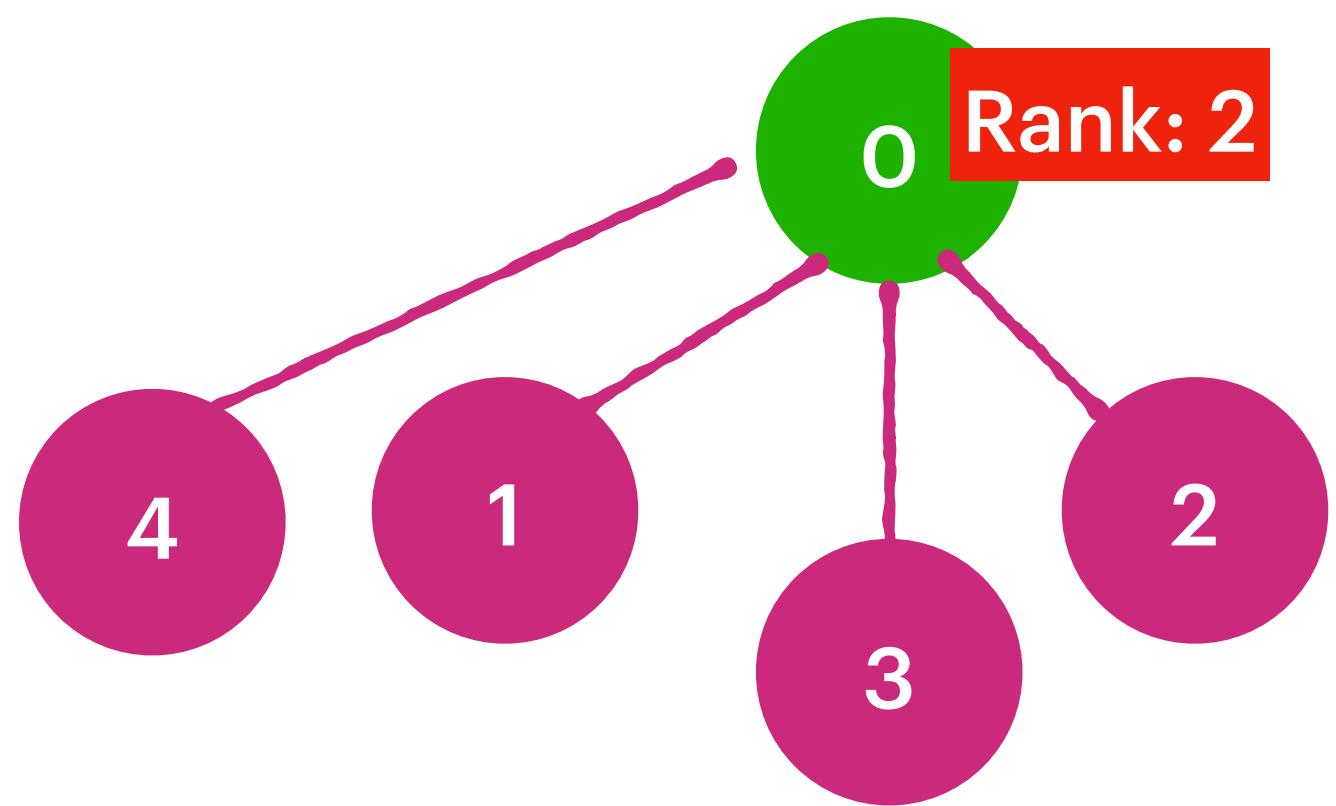
=====>



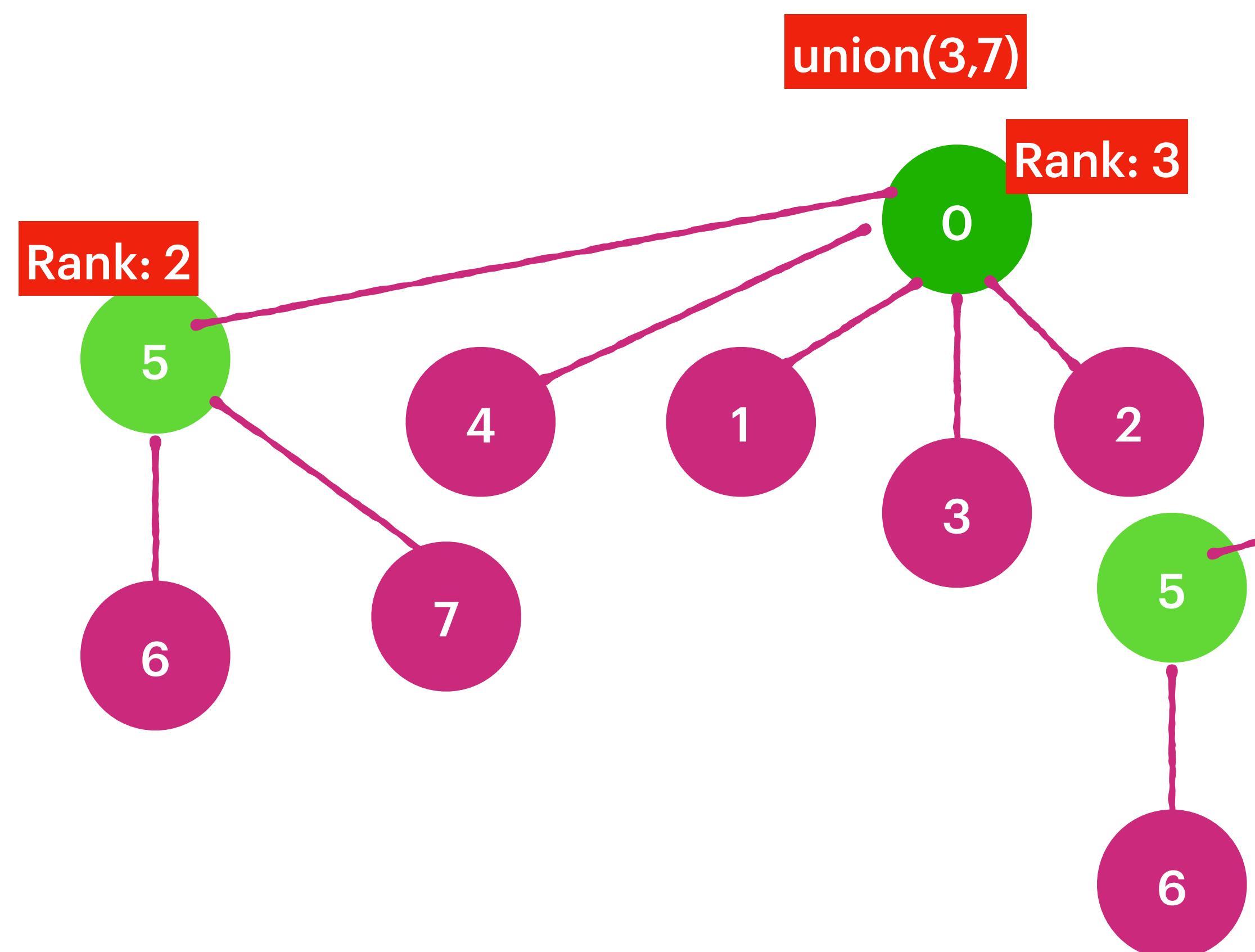


union(2,7)  
Find(2) = 0  
find(7) = 4  
root[4] = 0  
=====>



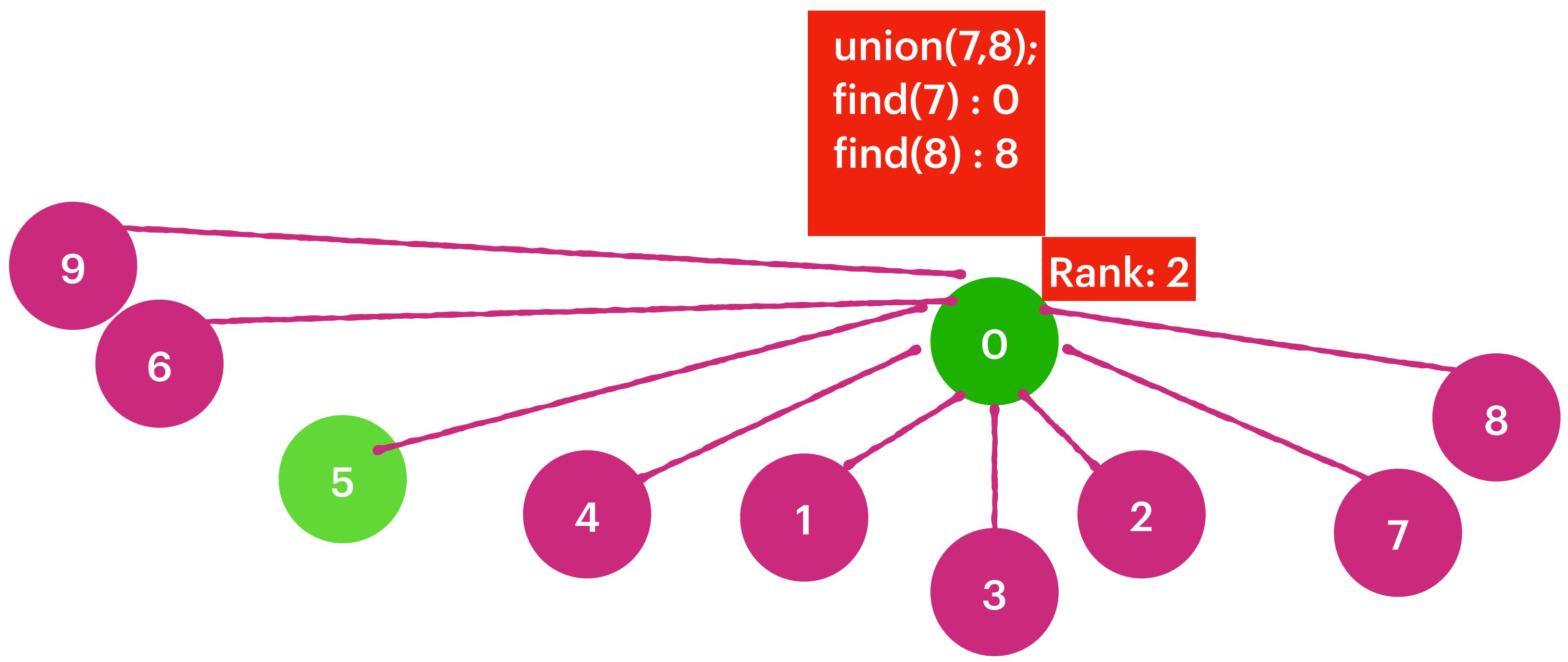


// 0-1-2-3-4 :: 5-6-7 :: 8



union(7,8);  
find(7) : 0  
find(8) : 8

```
union(7,8);  
find(7) : 0  
find(8) : 8
```



```
union(6,9)  
Find(6) : 0  
Find(9) : 9
```

## Graph Valid Tree

You have a graph of  $n$  nodes labeled from 0 to  $n - 1$ . You are given an integer  $n$  and a list of edges where  $\text{edges}[i] = [a_i, b_i]$  indicates that there is an undirected edge between nodes  $a_i$  and  $b_i$  in the graph.  
Return true if the edges of the given graph make up a valid tree, and false otherwise.

**Input:**  $n = 5$ ,  $\text{edges} = [[0,1],[0,2],[0,3],[1,4]]$

**Output:** true

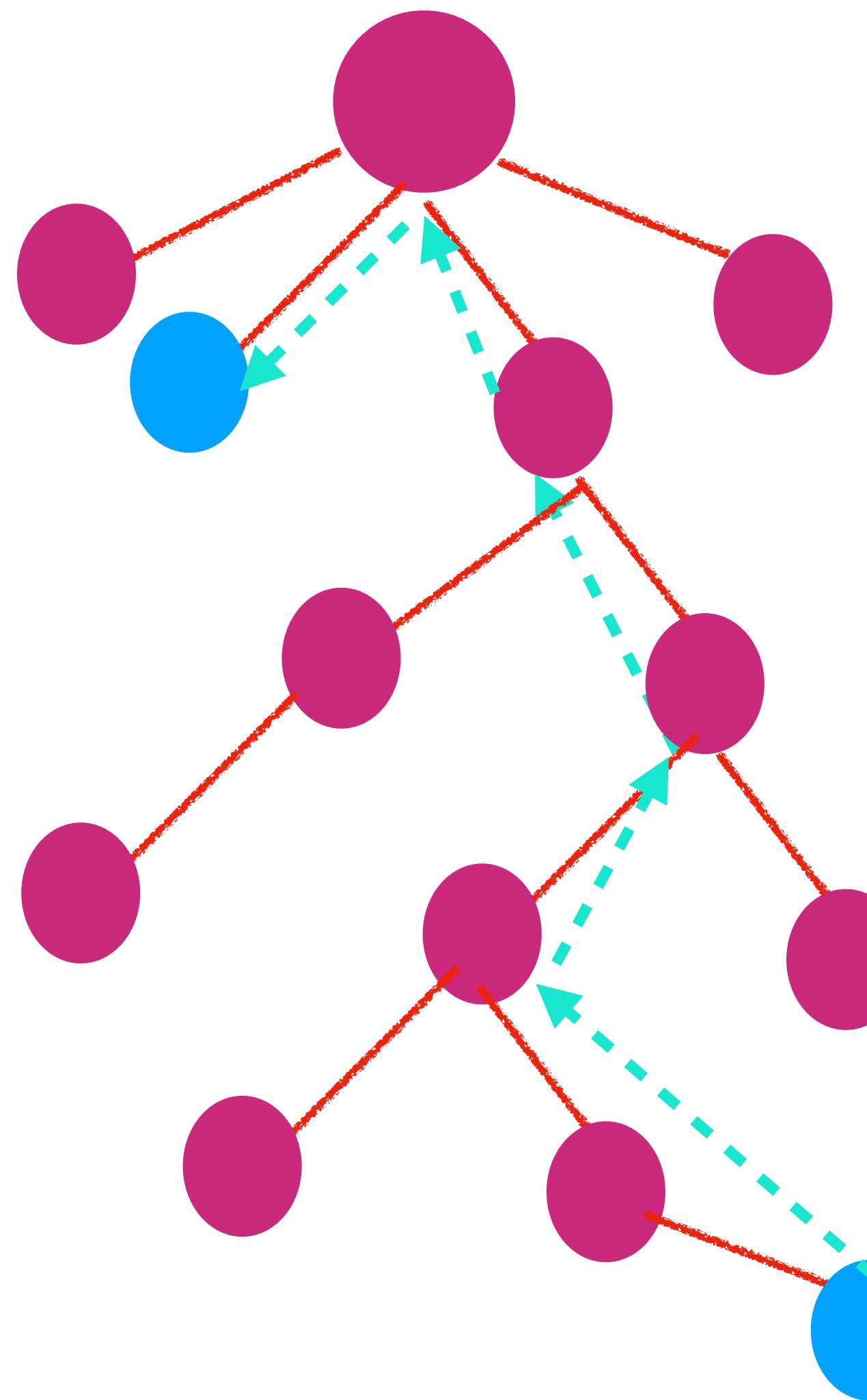
**Input:**  $n = 5$ ,  
 $\text{edges} = [[0,1],[0,4],[1,4],[2,3]]$   
**Output:** false

**Input:**  $n = 5$ ,  $\text{edges} = [[0,1],[1,2],[2,3],[1,3],[1,4]]$

**Output:** false

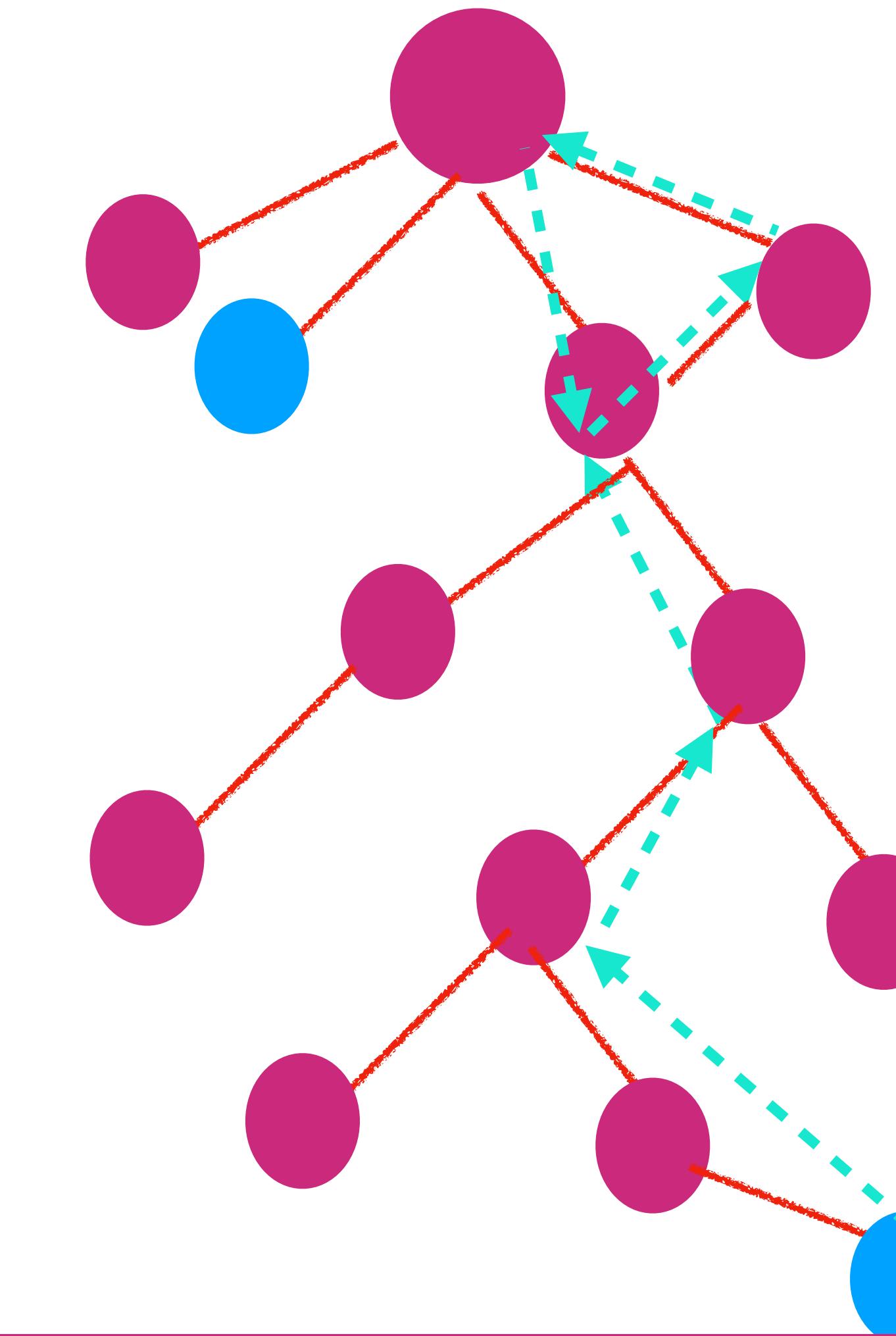
**Constraints :**  
 $1 \leq n \leq 2000$   
 $0 \leq \text{edges.length} \leq 5000$   
 $\text{edges}[i].length == 2$   
 $0 \leq a_i, b_i < n$   
 $a_i \neq b_i$  (No Self loop)

Graph with Valid Tree



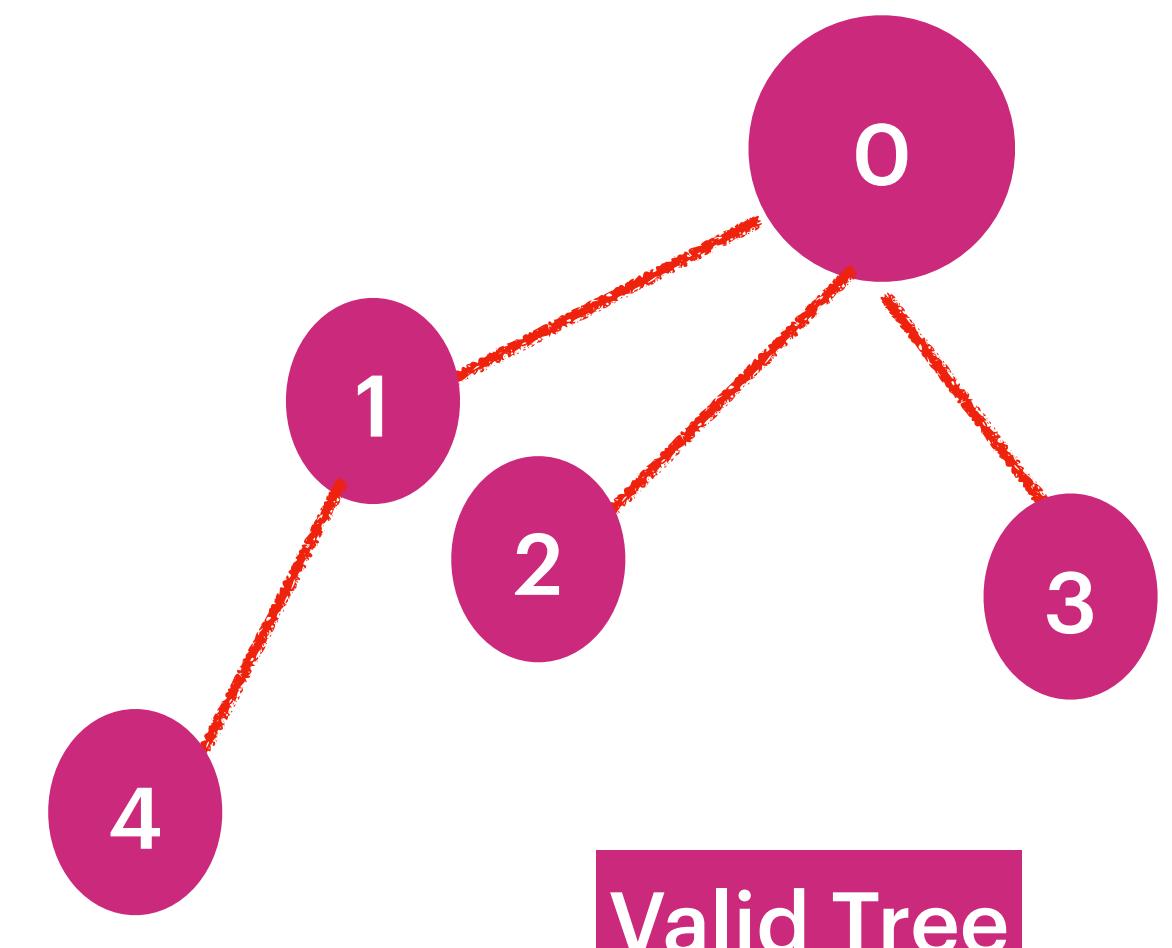
Its a valid tree because  
From one branch we can move to another branch

Graph with Invalid Tree

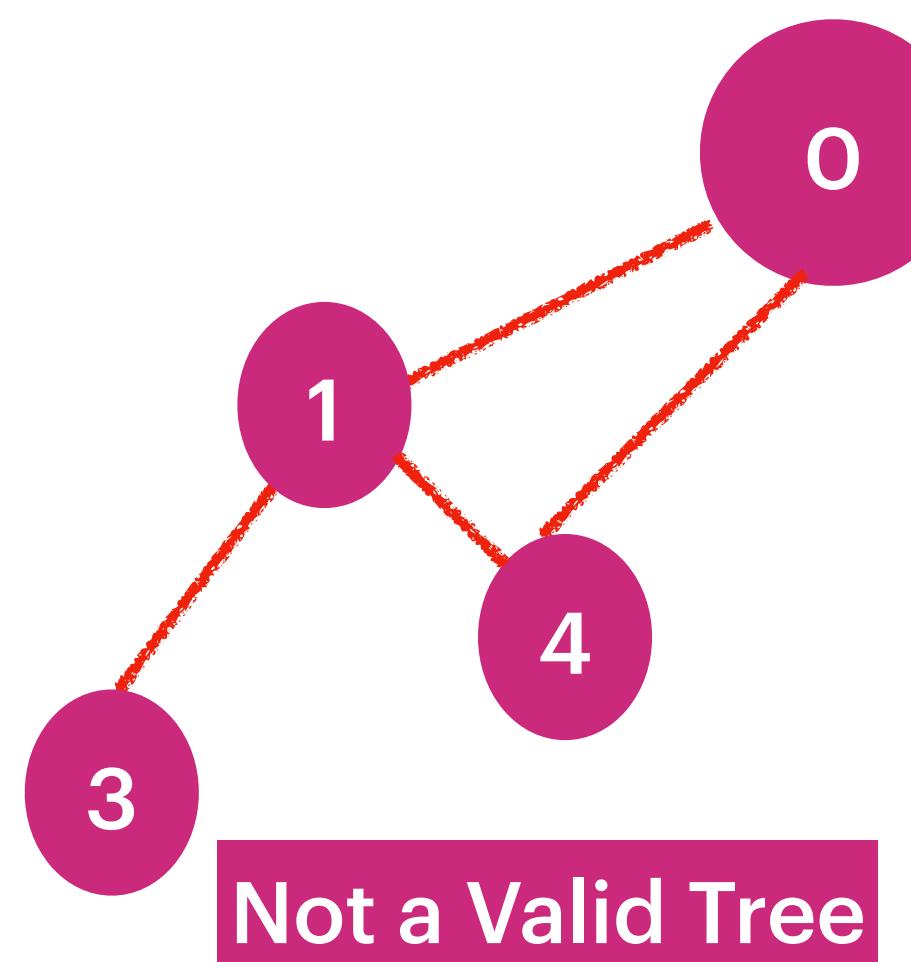


Its not a valid tree because  
, a child node has more than one immediate parent so it causes cycle.

**Input:**  $n = 5$ , edges = [[0,1],[0,2],[0,3],[1,4]]  
**Output:** true



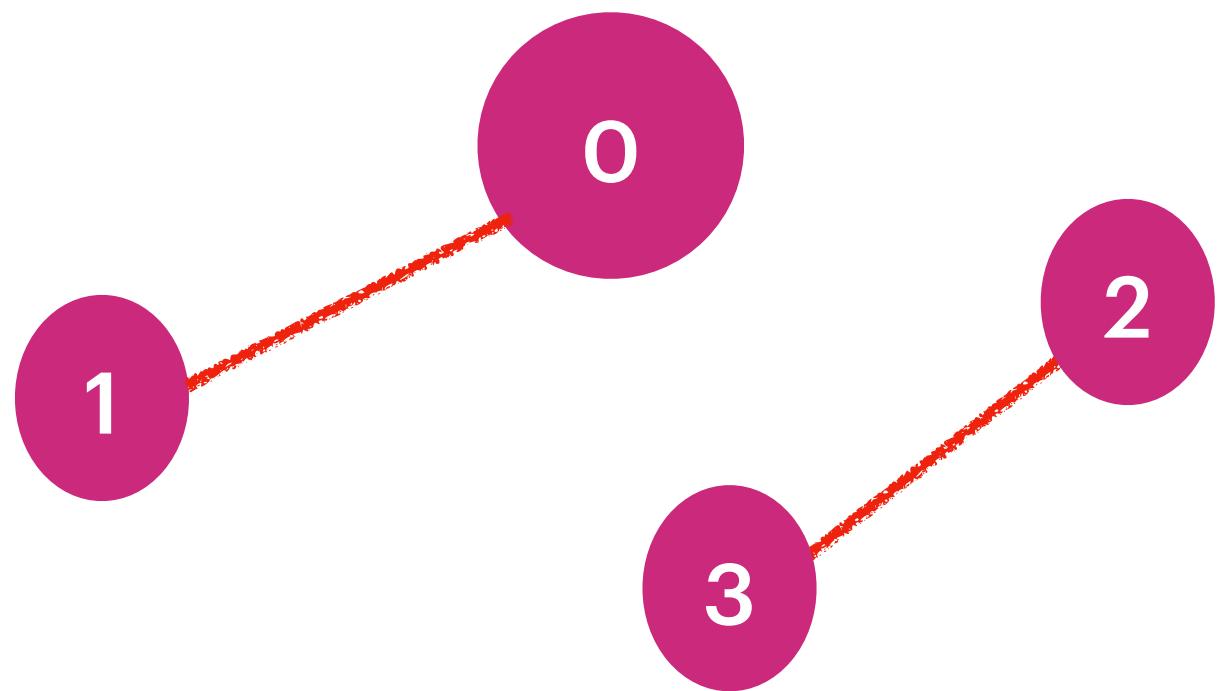
**Input:**  $n = 5$ ,  
edges = [[0,1],[0,4],[1,4],[1,3]]  
**Output:** false



**Input:**  $n = 5$ , edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]  
**Output:** false

For  $n$  vertexes we can have at max  $n-1$  edges.  
Here  $n = 5$  and edges = 5 so Not a Valid Tree.

n=4 [ [0,1], [2,3] ]



Its not a Valid Tree because

[[0,1],[0,4],[1,4],[1,3]] n = 5

0[0]  
1[1]  
2[2]  
3[3]  
4[4]

[0,1] => 0[0]-1[0]

[0,4] => 0[0]-1[0]-4[0]

[1,4] => 1[0] - 4[0] They are already connected : Its a loop return false

Input: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]

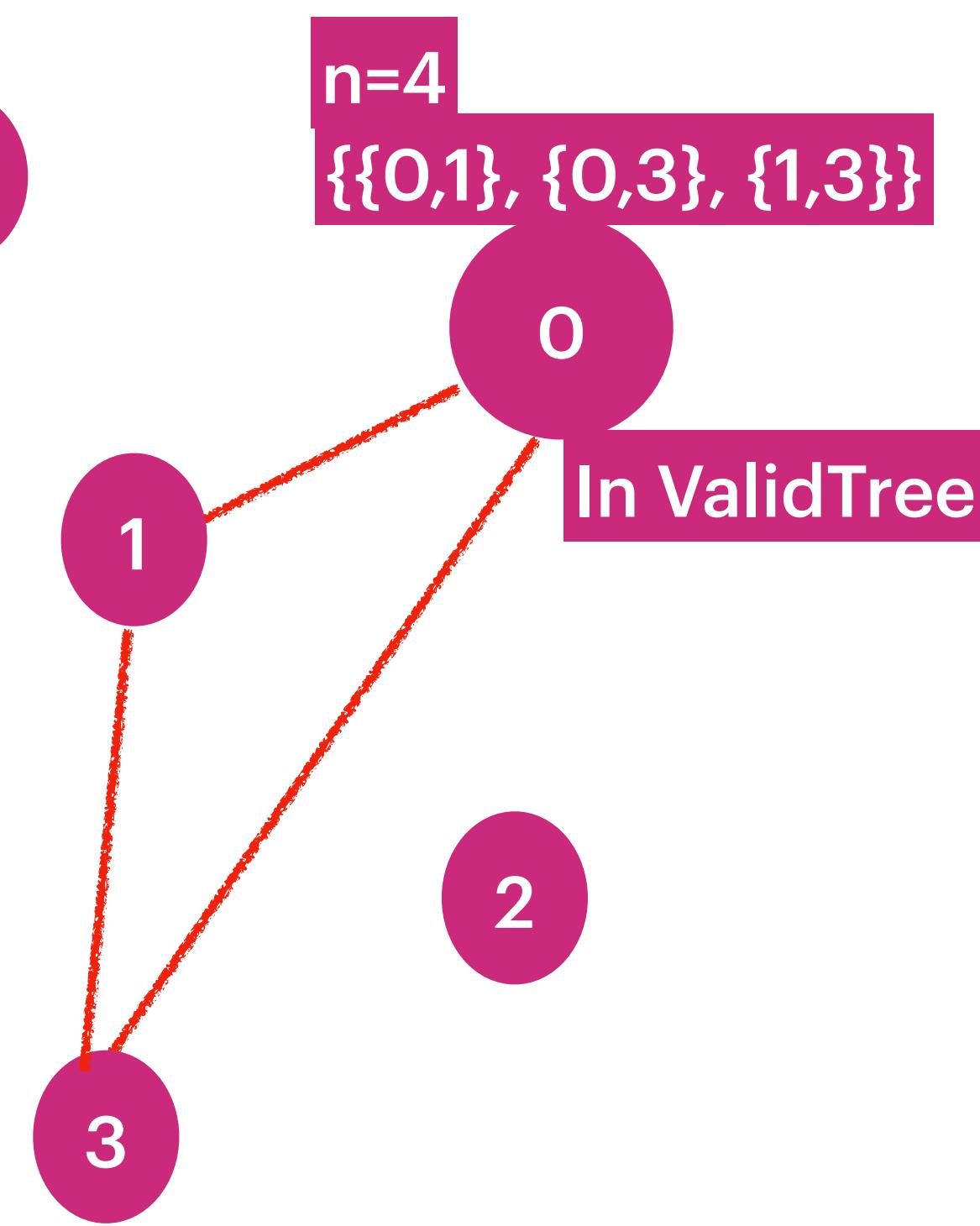
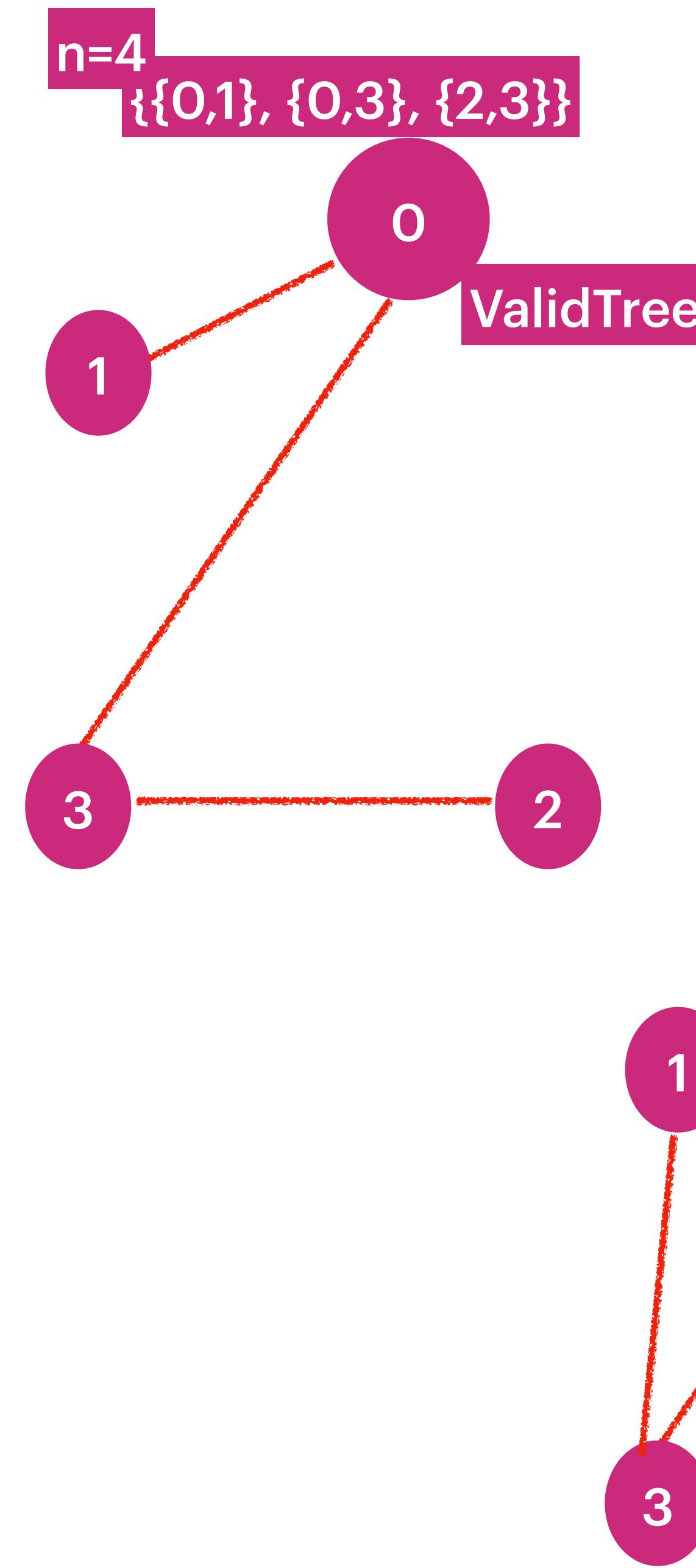
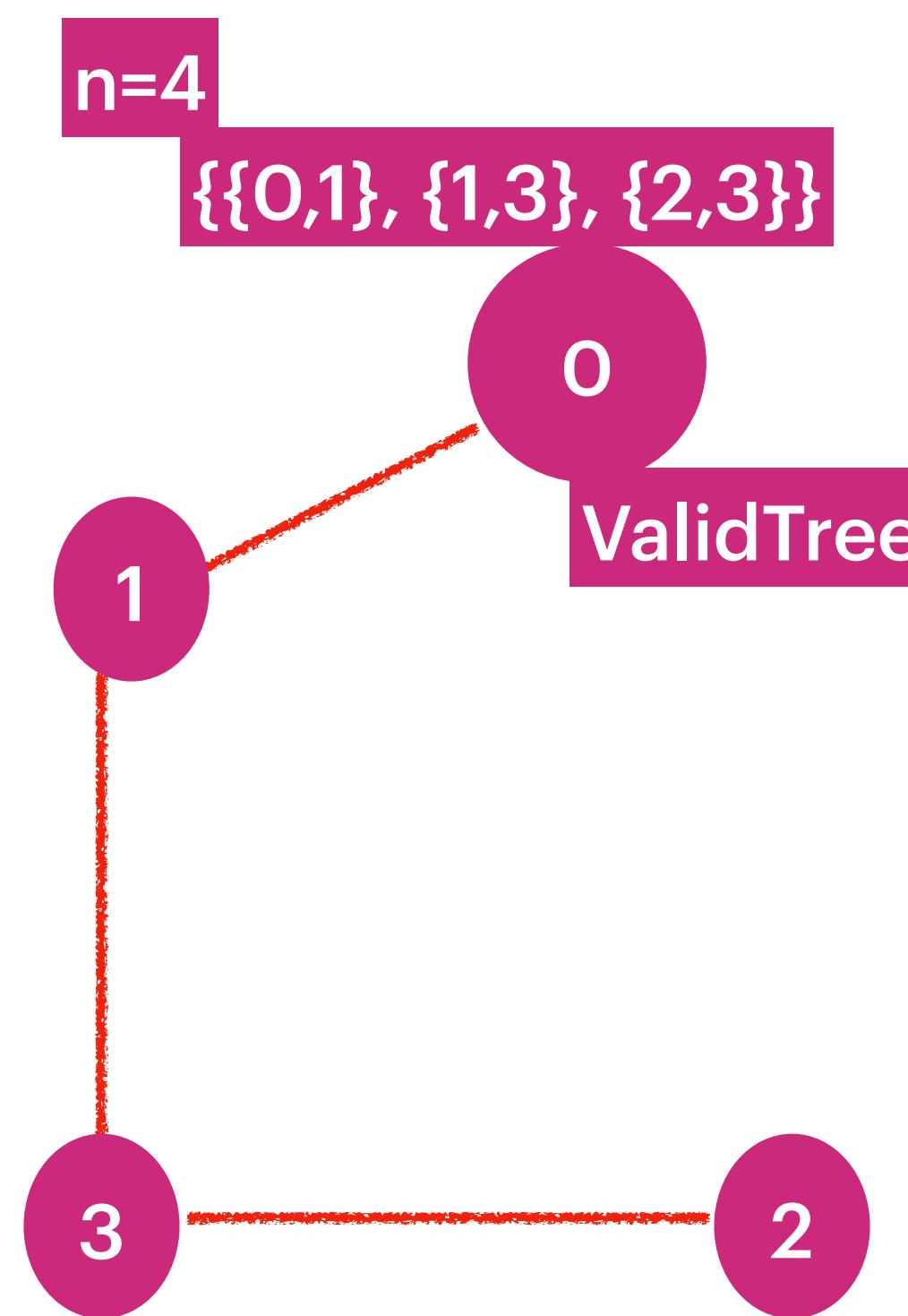
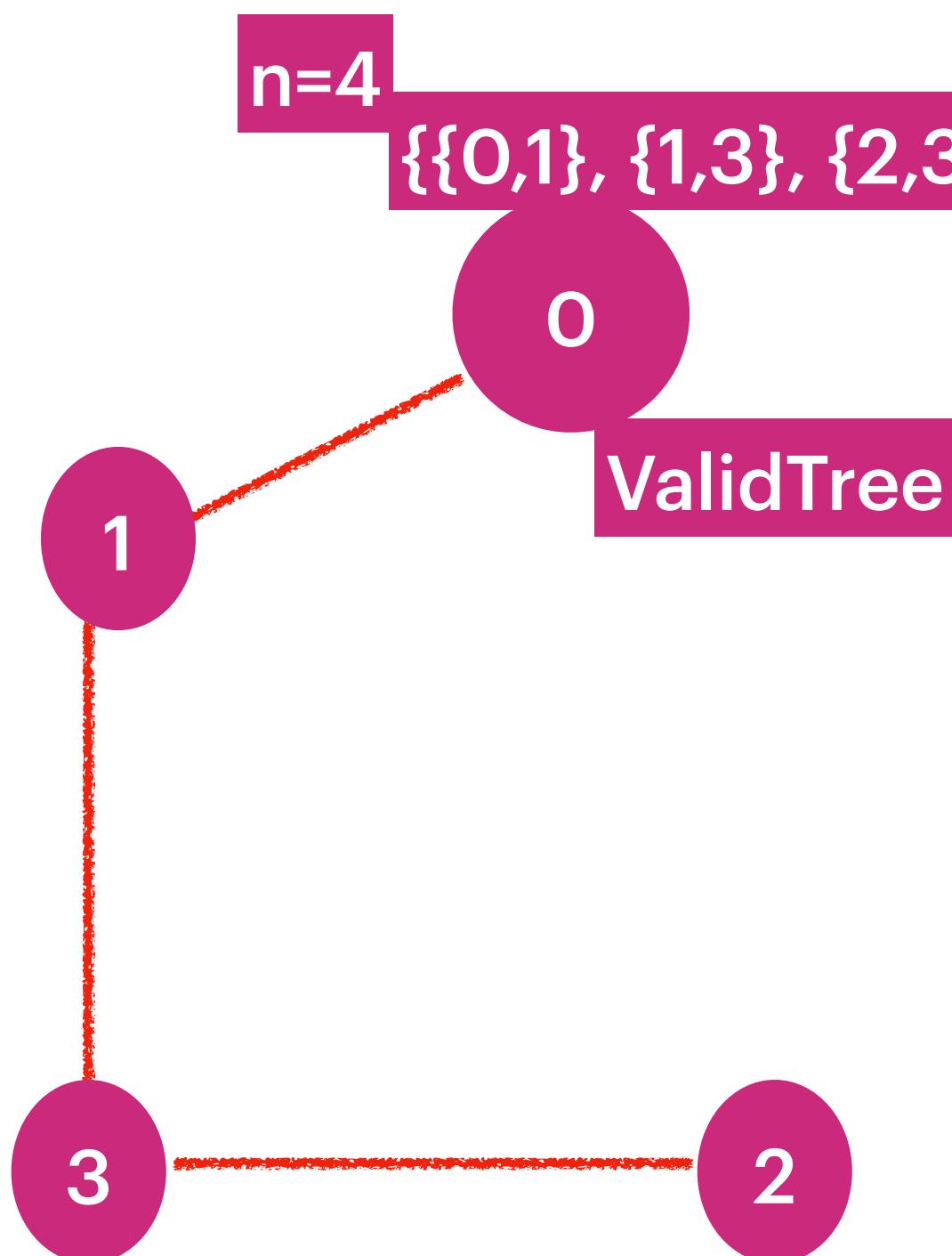
0[0]  
1[1]  
2[2]  
3[3]  
4[4]

[0,1] => 0[0]-1[0]

[0-2] => 0[0]-1[0]- 2[0]

[0-3] => 0[0]-1[0]- 2[0]-3[0]

[1-4] =>0[0]-1[0]- 2[0]-3[0]-4[0] Its a valid Tree : return true



## Number of Connected Components in an Undirected Graph

You have a graph of n nodes. You are given an integer n and an array edges where edges[i] = [ai, bi] indicates that there is an edge between ai and bi in the graph.

Return the number of connected components in the graph.

**Input:** n = 5, edges = [[0,1],[1,2],[3,4]]

**Output:** 2

**Input:** n = 5, edges = [[0,1],[1,2],[2,3],[3,4]]

**Output:** 1

**Constraints :** 1 <= n <= 2000

1 <= edges.length <= 5000

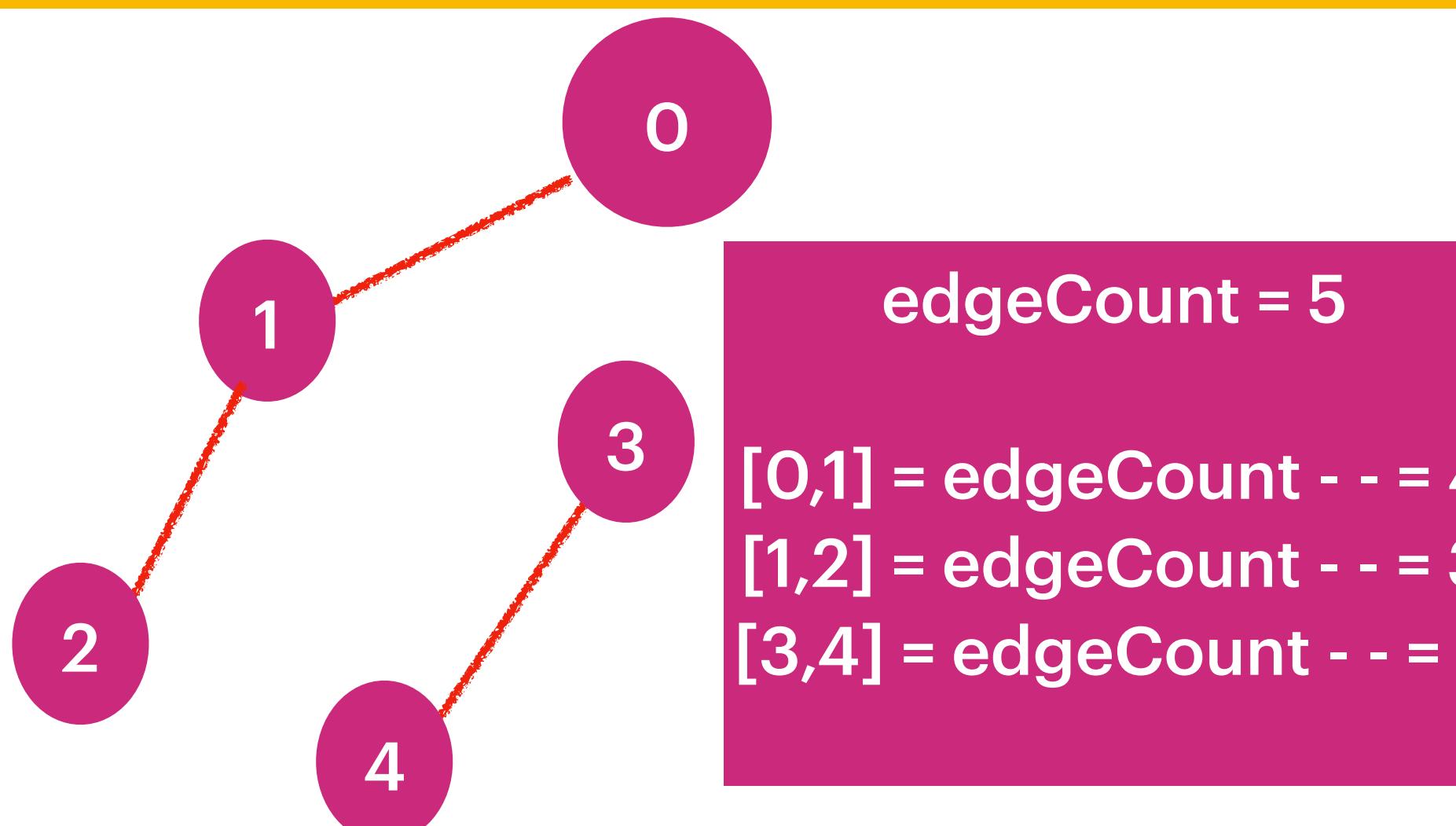
edges[i].length == 2

0 <= ai <= bi < n

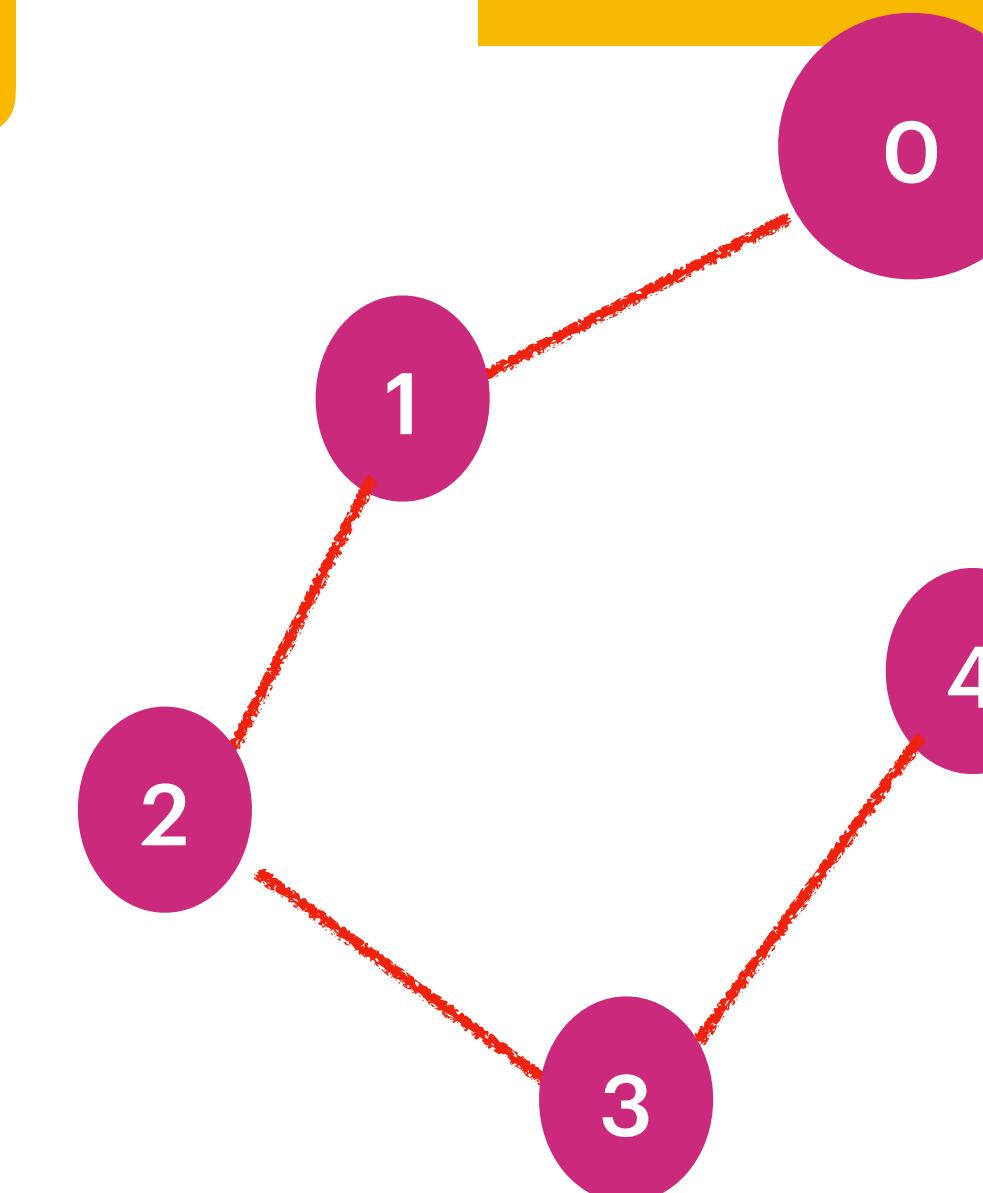
ai != bi

There are no repeated edges.

**Input: n = 5, edges = [[0,1],[1,2],[3,4]]**  
**Output: 2**

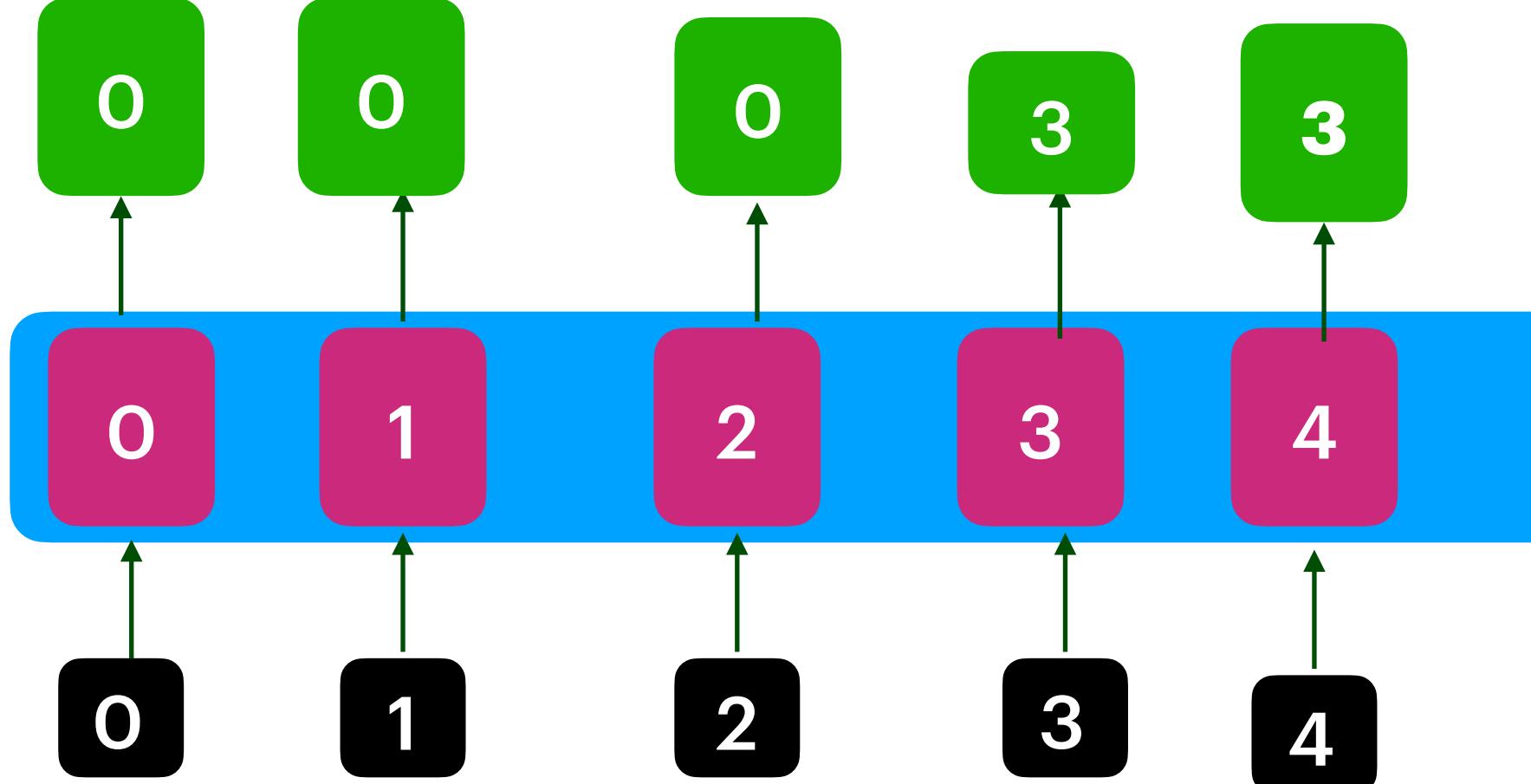


**Input: n = 5, edges = [[0,1],[1,2],[2,3],[3,4]]**  
**Output: 1**

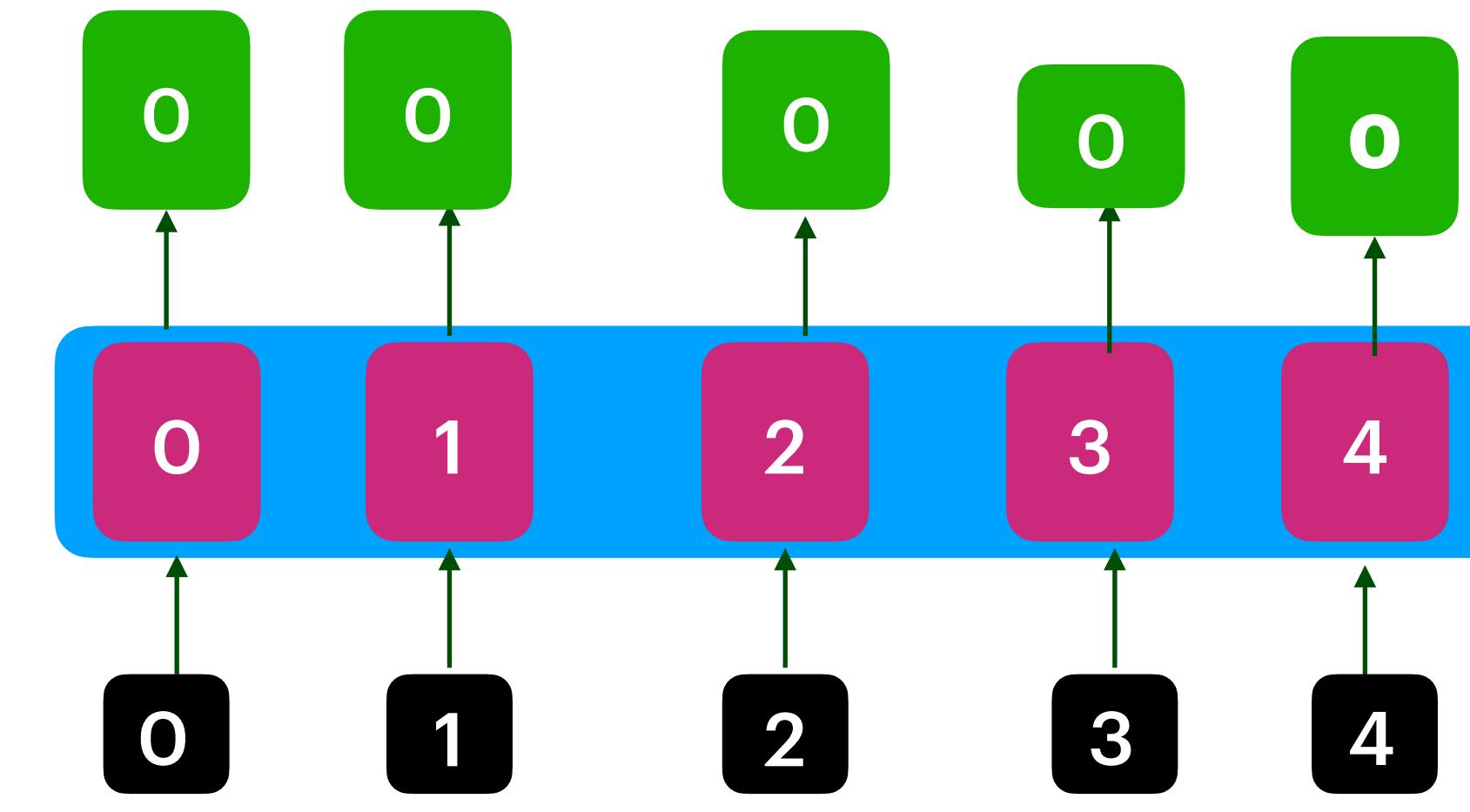


**edgeCount = n = 5**  
**edgeCount**  
[0,1] = edgeCount- - = 4  
[1,2] = edgeCount- - = 3  
[2,3] = edgeCount- - = 2  
[3,4] = edgeCount- - = 1

**Connected Components / Paths. = 2**



**Connected Components / Paths. = 1**



## Number of Provinces

There are n cities. Some of them are connected, while some are not. If city a is connected directly to city b, and city b is connected directly with city c, then city a is connected indirectly with city c. A province is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an  $n \times n$  matrix isConnected where isConnected[i][j] = 1 if the ith city and the jth city are directly connected, and isConnected[i][j] = 0 otherwise.

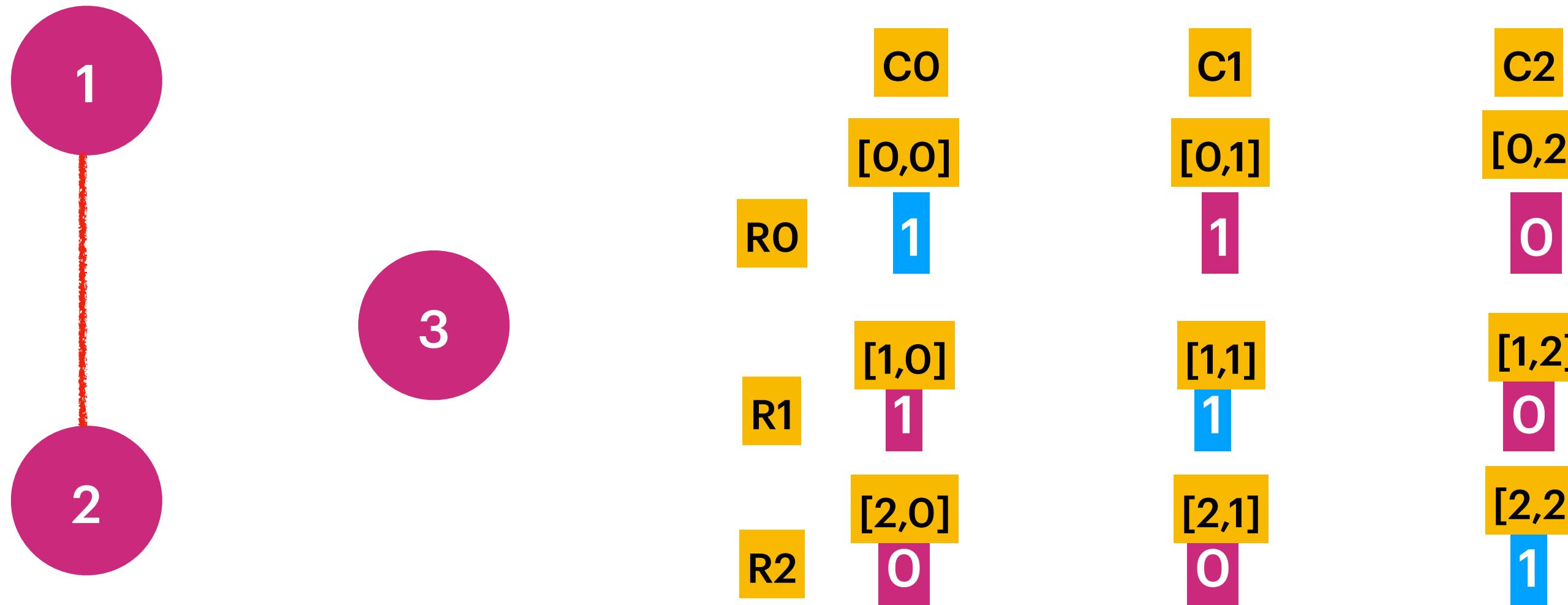
Return the total number of provinces.

**Input:** isConnected = [[1,1,0],[1,1,0],[0,0,1]]  
**Output:** 2

**Input:** isConnected = [[1,0,0],[0,1,0],[0,0,1]]  
**Output:** 3

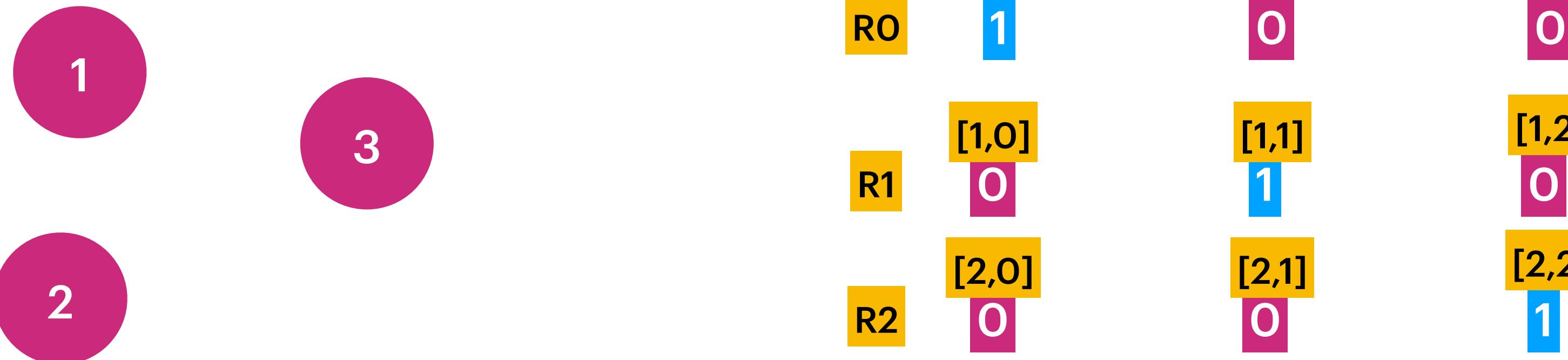
**Input:** isConnected = [[1,1,0],[1,1,0],[0,0,1]]  
**Output:** 2

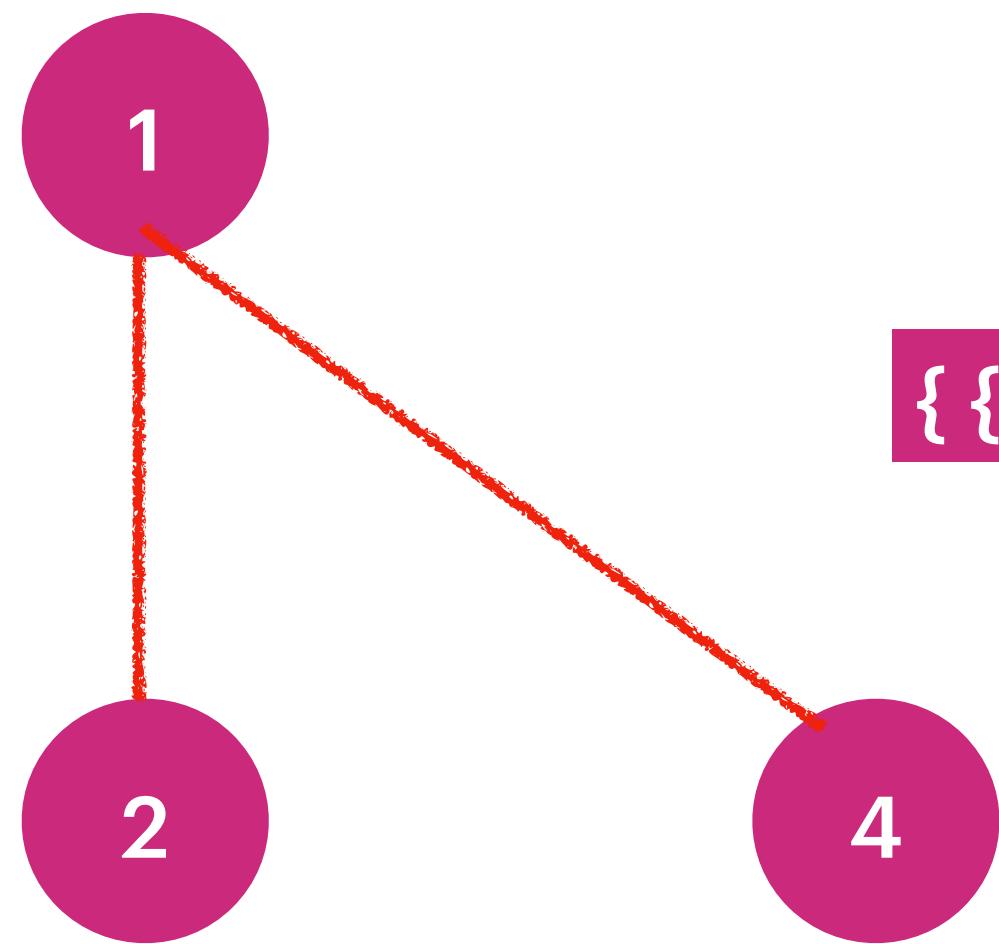
Count = 3  
Count - - = 2



**Input:** isConnected = [[1,0,0],[0,1,0],[0,0,1]]  
**Output:** 3

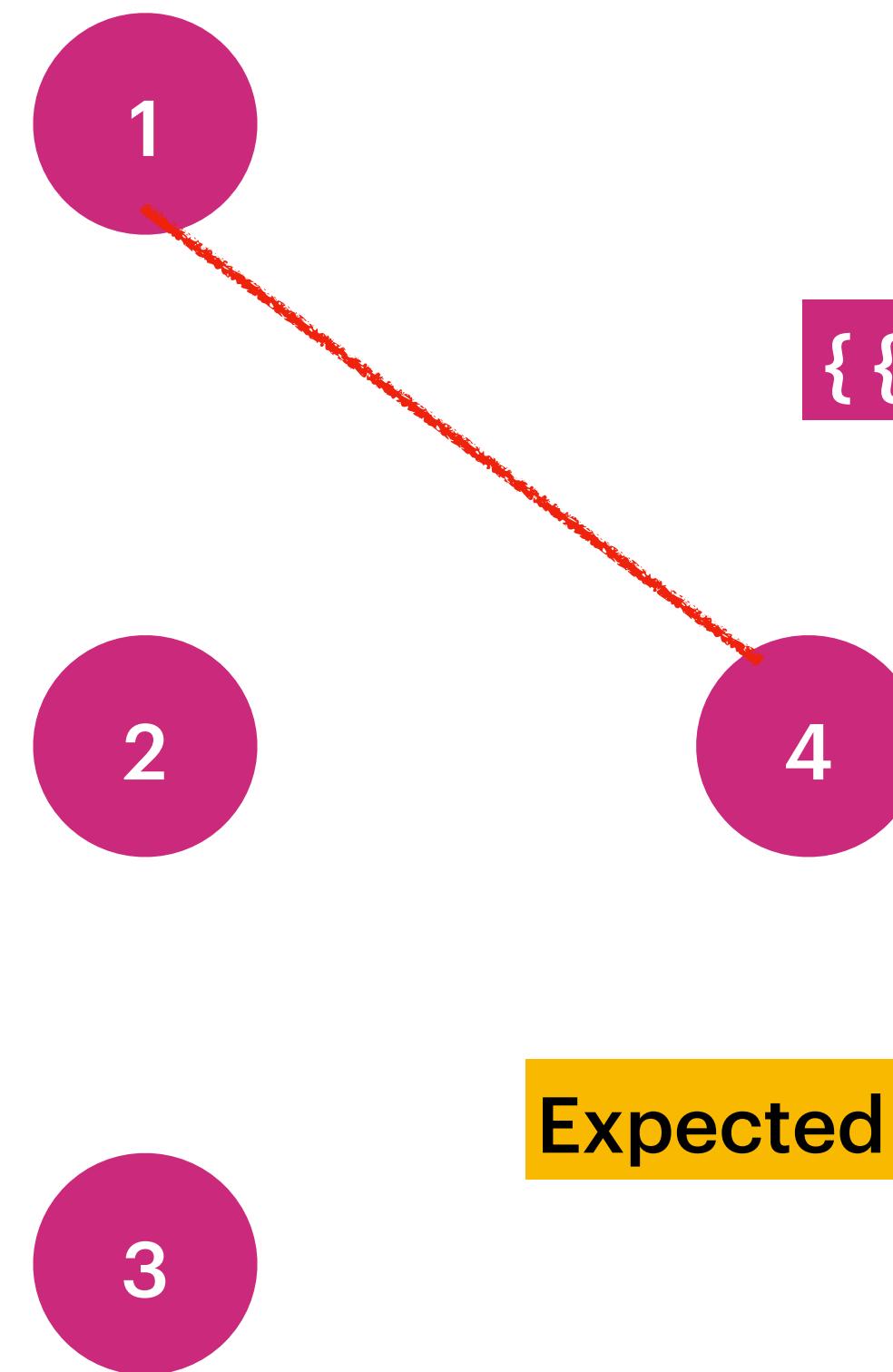
Count = 3  
Count - - = 2





{ {1,1,0,1}, {1,1,0,0},{0,0,1,0},{1,0,0,1} }

Expected Output : 2



{ {1,0,0,1}, {0,1,0,0},{0,0,1,0},{1,0,0,1} }

Expected Output : 3

## The Earliest Moment When Everyone Become Friends in Instagram

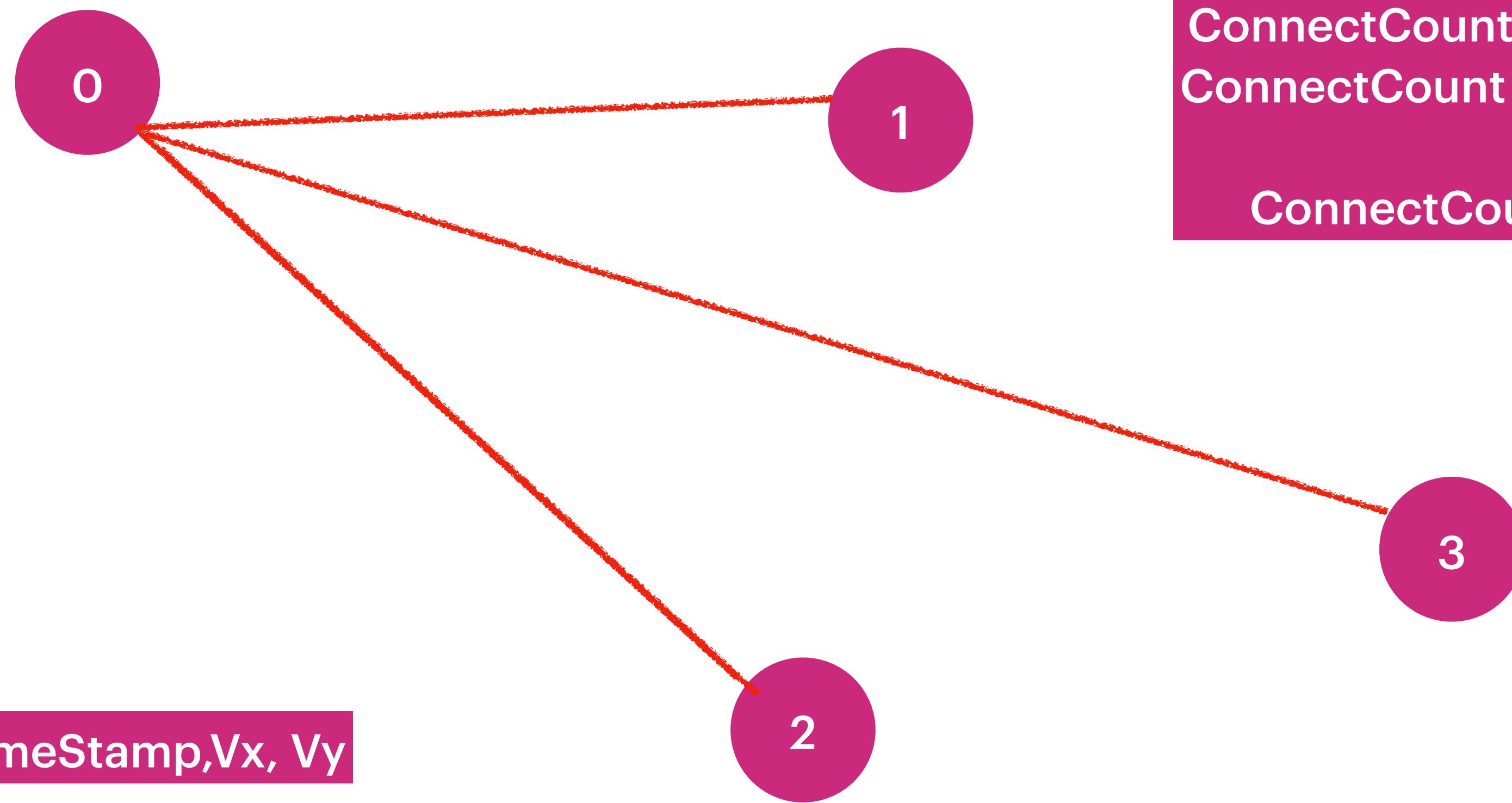
There are  $n$  people in a social group labeled from 0 to  $n - 1$ . You are given an array  $\text{logs}$  where  $\text{logs}[i] = [\text{timestamp}_i, x_i, y_i]$  indicates that  $x_i$  and  $y_i$  will be friends at the time  $\text{timestamp}_i$ .

Friendship is symmetric. That means if  $a$  is friends with  $b$ , then  $b$  is friends with  $a$ . Also, person  $a$  is acquainted with a person  $b$  if  $a$  is friends with  $b$ , or  $a$  is a friend of someone acquainted with  $b$ . Return the earliest time for which every person became acquainted with every other person. If there is no such earliest time, return -1.

**Input:**  $\text{logs} = [[20190101,0,1],[20190104,3,4],[20190107,2,3],[20190211,1,5],[20190224,2,4],[20190301,0,3],[20190312,1,2],[20190322,4,5]]$ ,  $n = 6$   
**Output:** 20190301

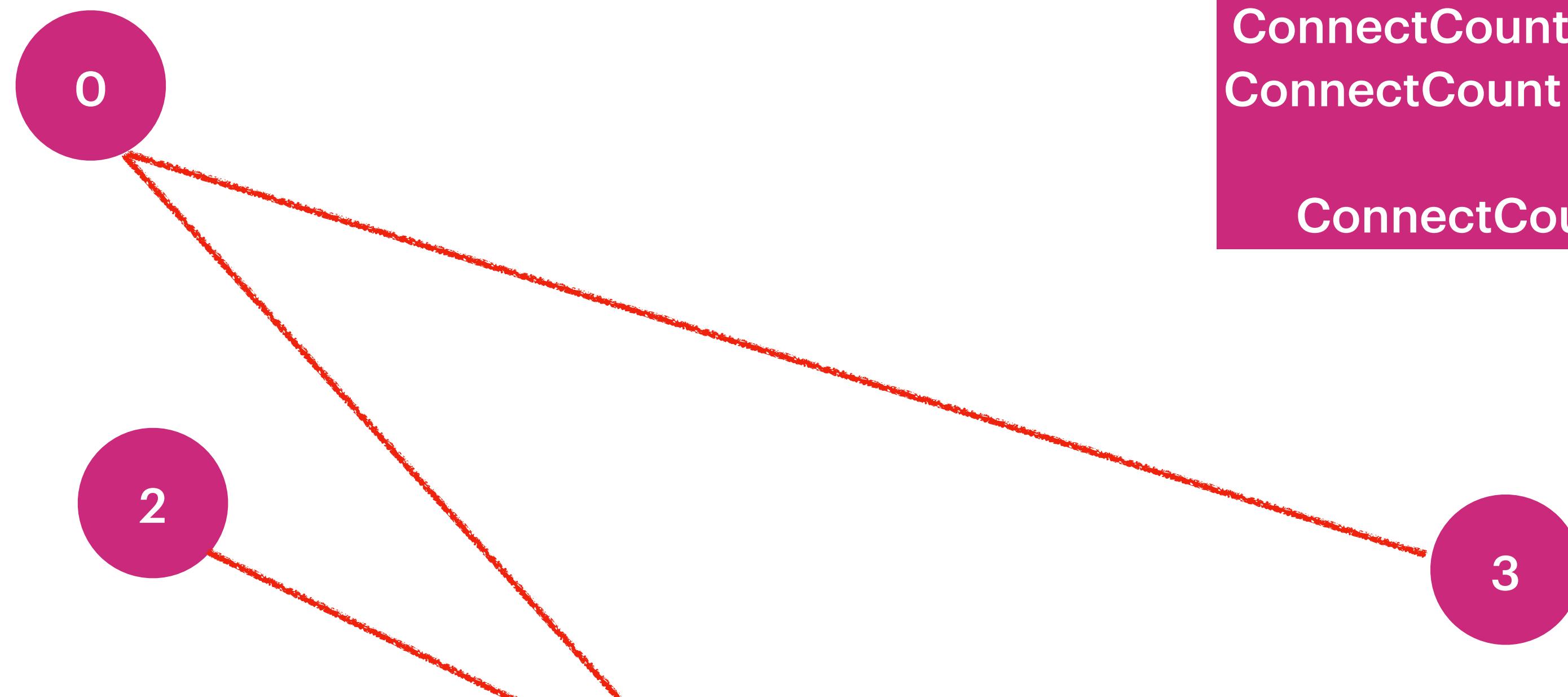
**Input:**  $\text{logs} = [[0,2,0],[1,0,1],[3,0,3],[4,1,2],[7,3,1]]$ ,  $n = 4$   
**Output:** 3

$[[9,3,0],[0,2,1],[8,0,1],[1,3,2],[2,2,0],[3,3,1]]$



ConnectCount = n=4 = 3  
ConnectCount = n-1 stop  
ConnectCount = 0

**Input:** logs = [[0,2,0],[1,0,1],[3,0,3],[4,1,2],[7,3,1]], n = 4  
**Output:** 3



ConnectCount = n=4 = 3  
ConnectCount = n-1 stop  
ConnectCount = 0

TimeStamp,Vx, Vy

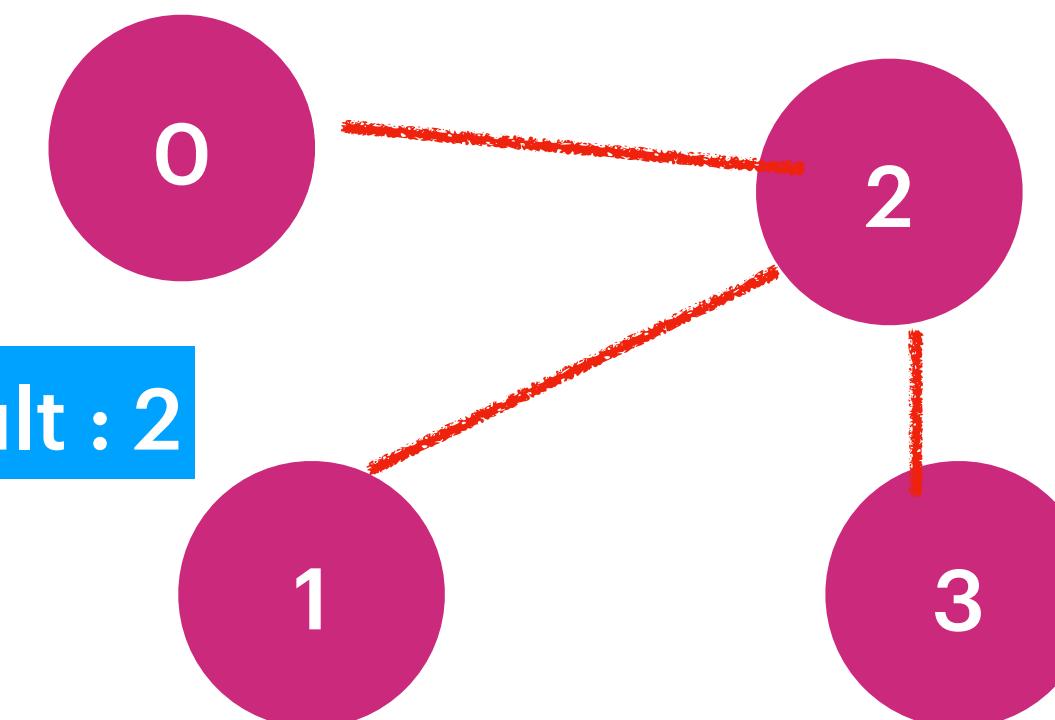
Not Expected Result : 8

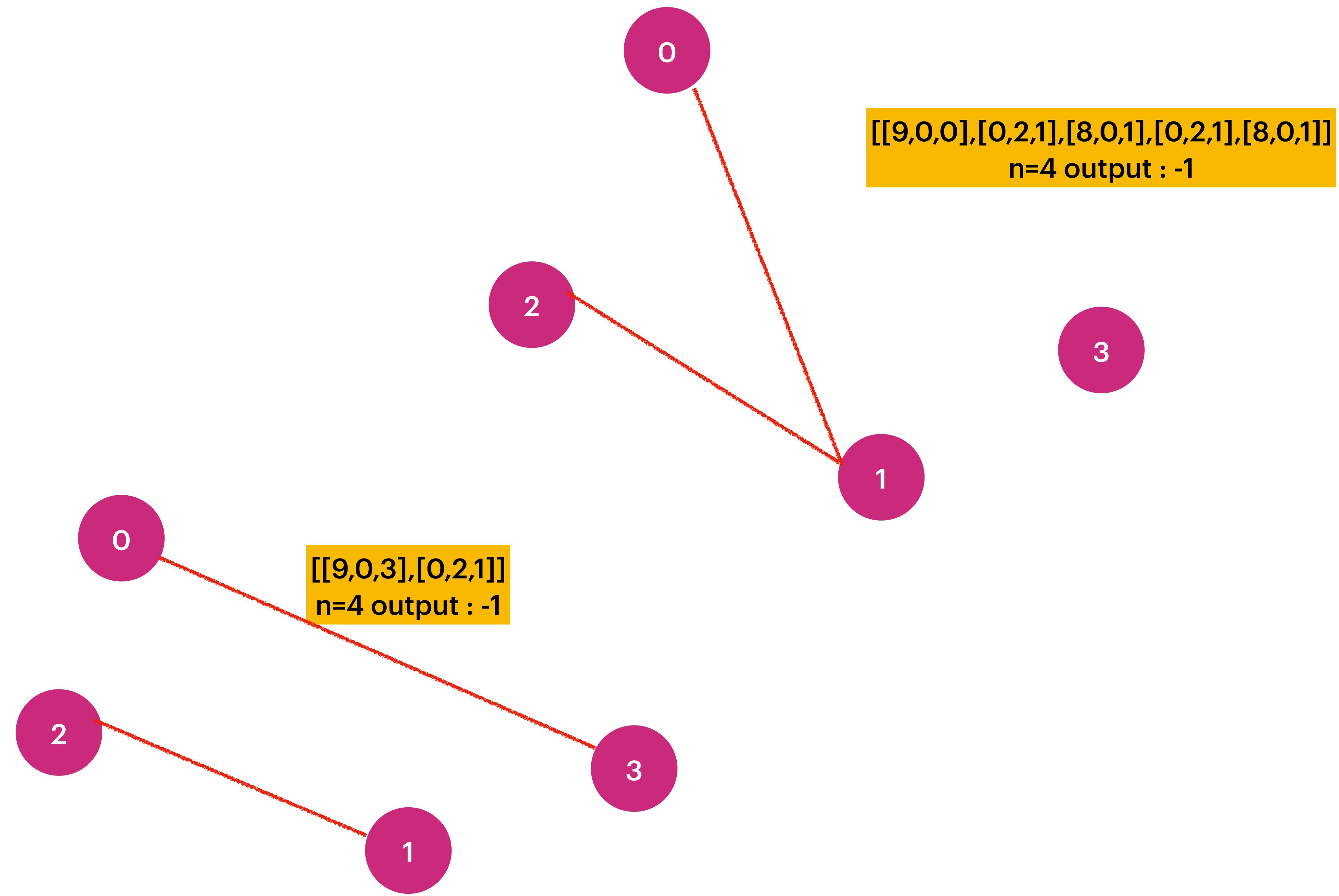
**[[9,3,0],[0,2,1],[8,0,1],[1,3,2],[2,2,0],[3,3,1]]**  
n=4 output : 2

**[[0,2,1],[1,3,2],[2,2,0],[3,3,1],[8,0,1],[9,3,0]]**  
Output : 2

TreeMap<Key(timeStamp),value[array]>

True Expected Result : 2





Design Graph

AddVertex

AddEdge

RemoveEdge

Remove Vertex

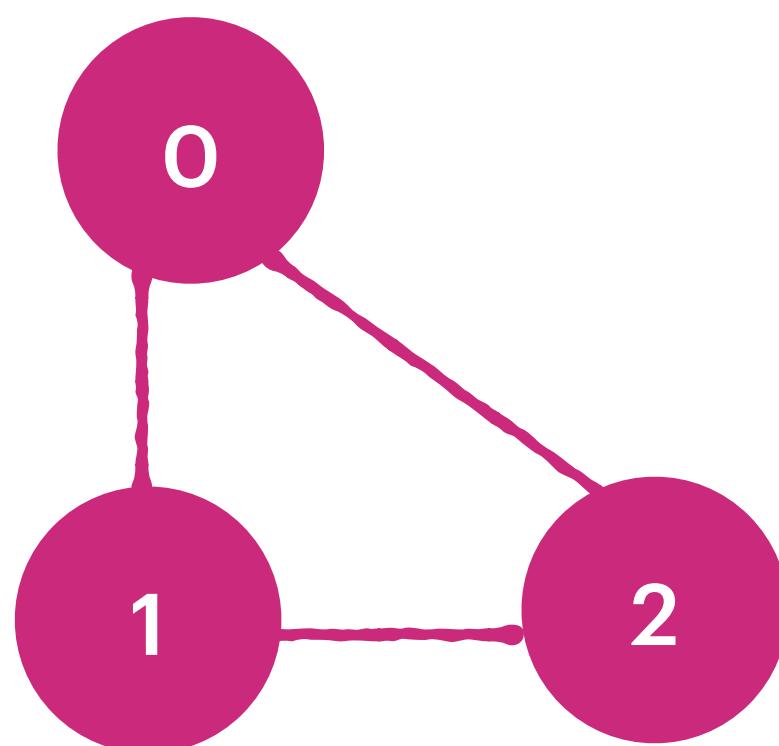
Print

$n = 3, \text{edges} = [[0,1],[0,2],[1,2],[2,1],[2,0],[1,0]]$

$O(1)$

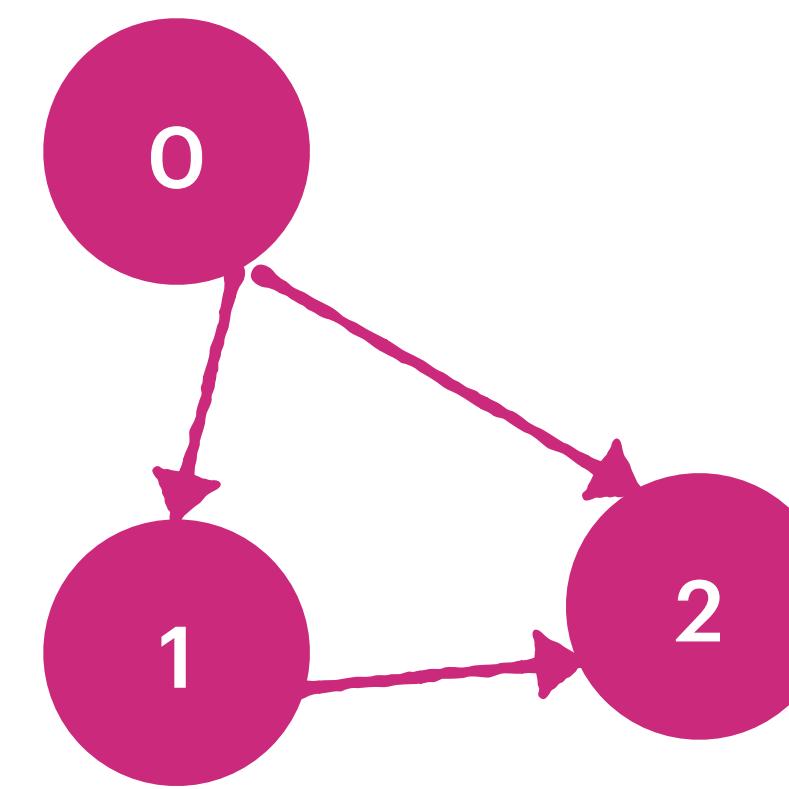
$O(V)$

$O(V+E)$

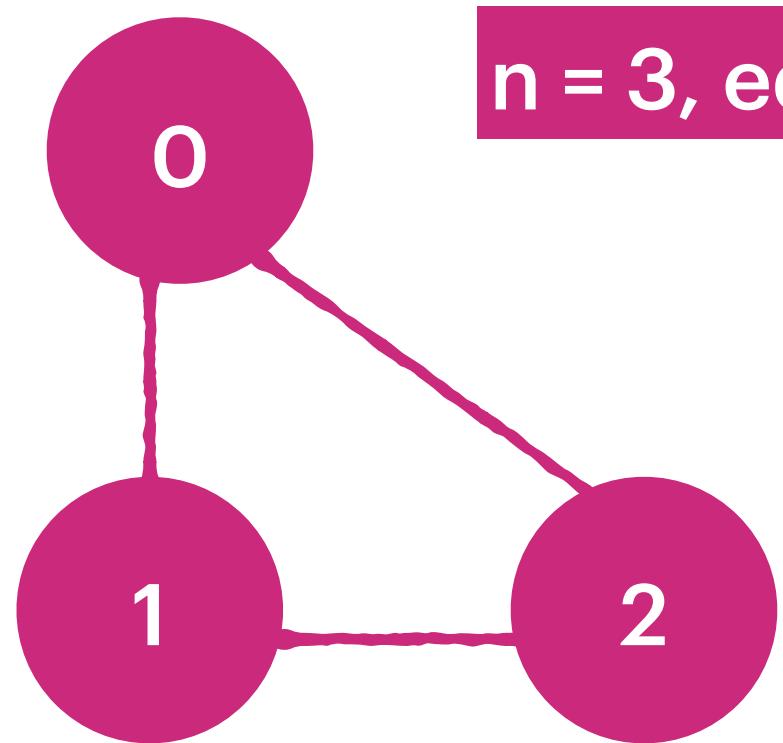


UnDirected Graph

$n = 3, \text{edges} = [[0,1],[1,2],[0,2]]$



Directed Graph

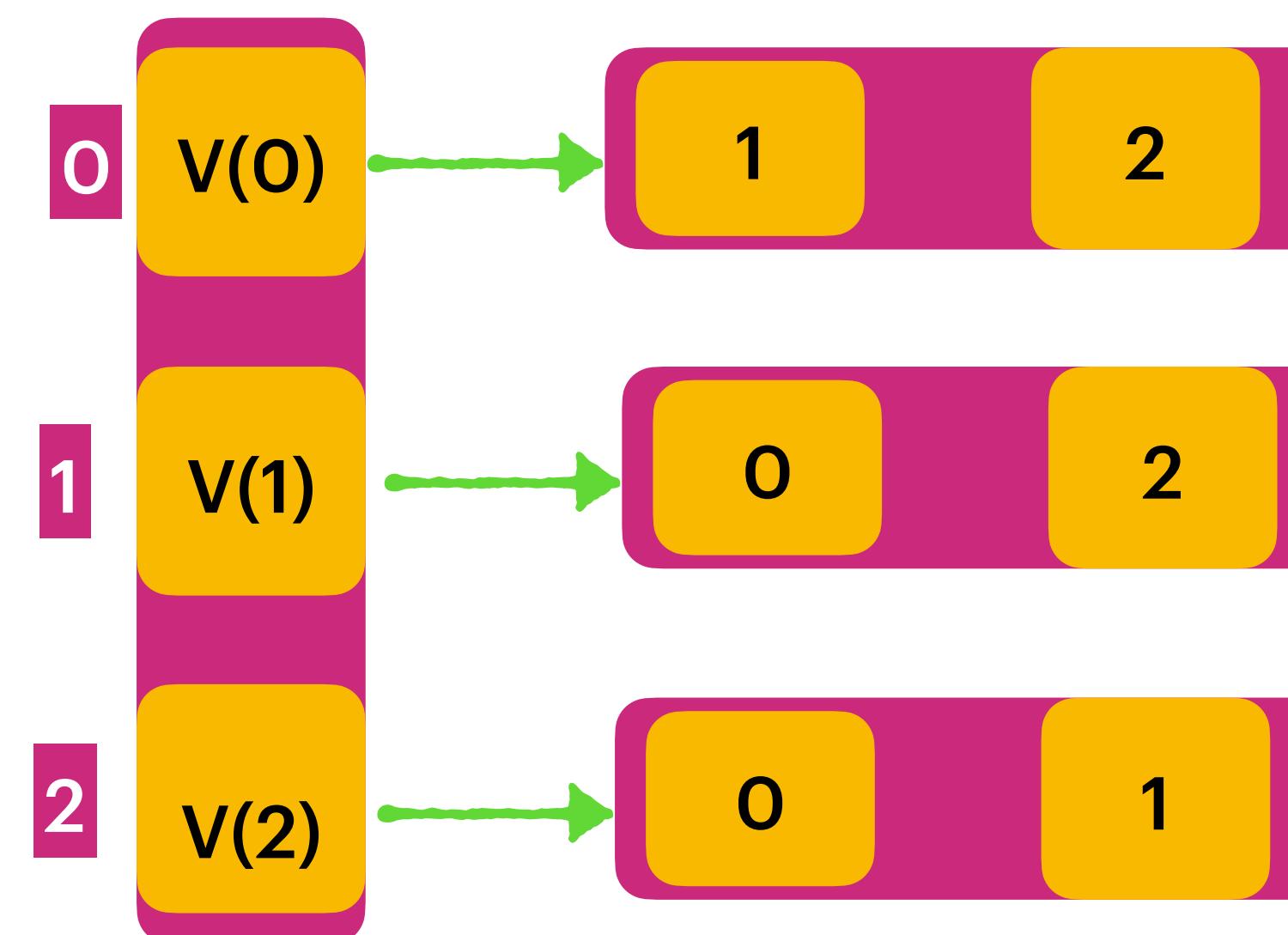


$n = 3$ , edges = [[0,1],[0,2],[1,2],[2,1],[2,0],[1,0]]

UnDirected Graph

List< List<Integer> >

Adjacent List ::



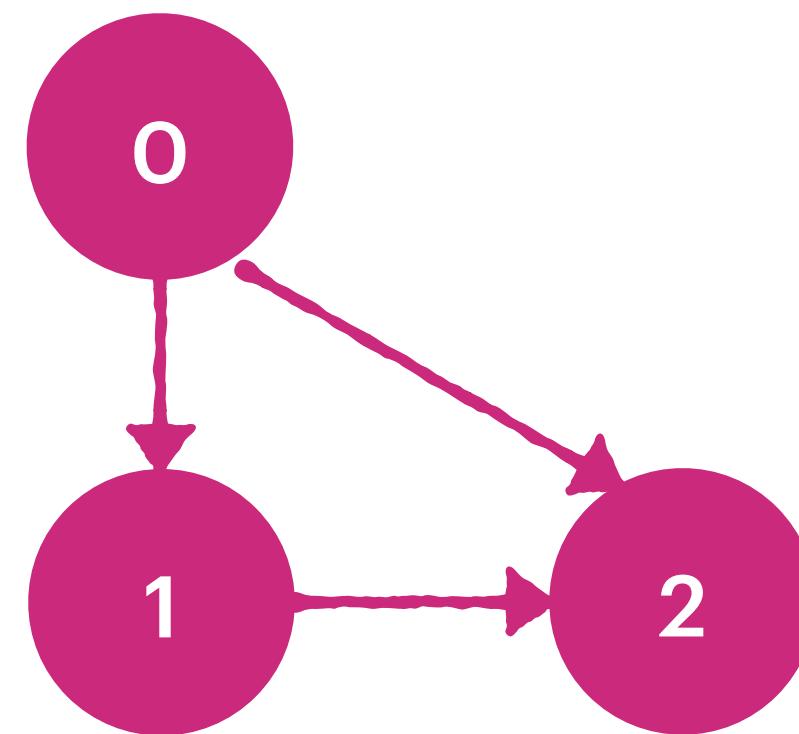
Matrix Row represents vertex

Matrix Column represents connections to the vertexes

Adjacent Matrix:  
 $n \times n$

0	1	2
0	0	1
1	1	0
2	1	1

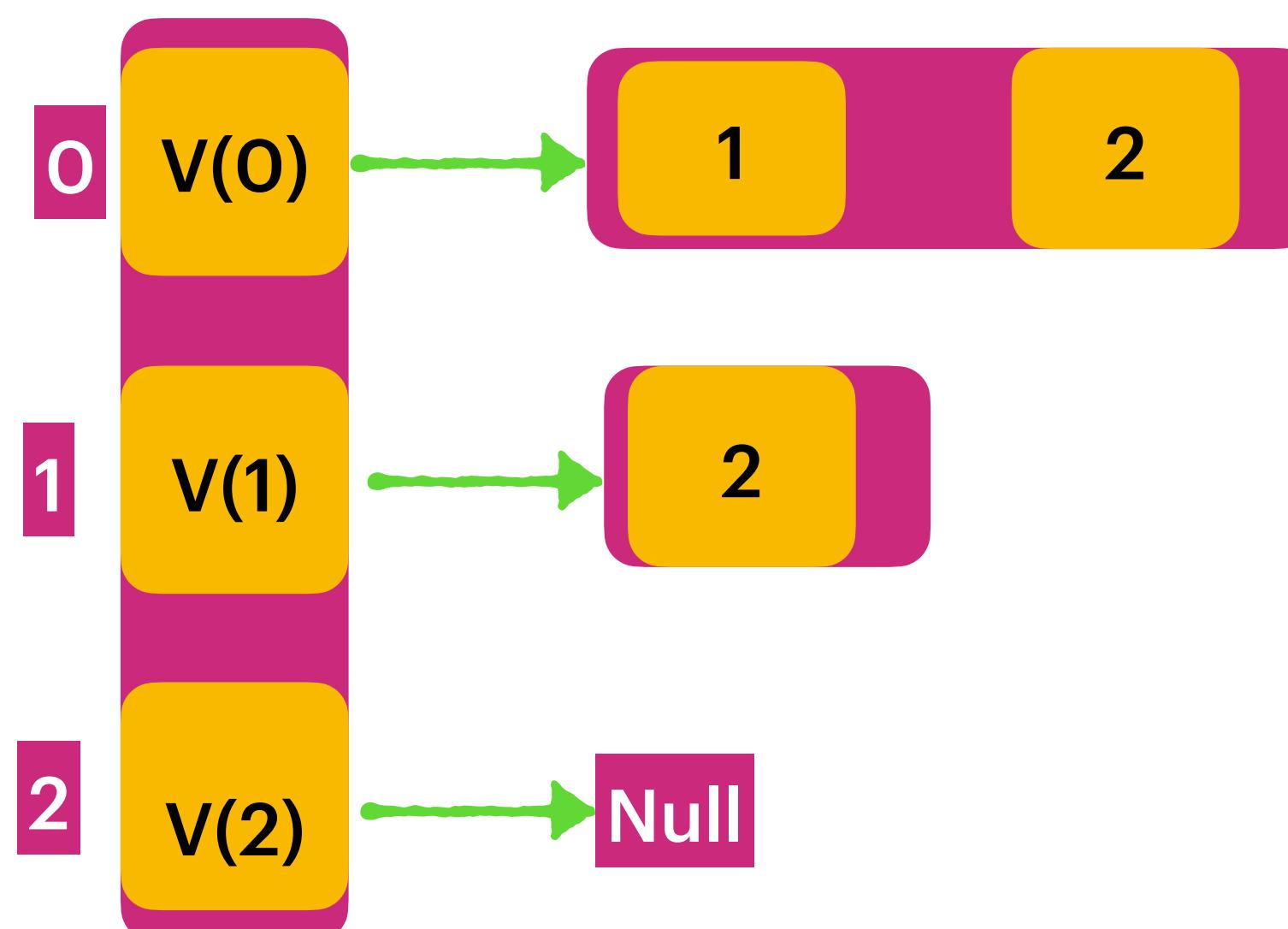
$n = 3$ , edges = [[0,1],[1,2],[0,2]]



Directed Graph

List< List<Integer> >

Adjacent List ::



Matrix Row represents vertex

Matrix Column represents connections to the vertexes

Adjacent Matrix:  
 $n \times n$

0	1	2
0	1	1
0	0	1
0	0	0

## Adjacent List

Add Vertex

$O(1)$

Remove Vertex

$O(V+E)$

Add Edge

$O(1)$

Remove Edge

$O(E)$

## Adjacent Matrix

$O(n^2)$

$O(n^2)$

$O(1)$

$O(1)$

## Adjacent List

Add Vertex

$O(1)$

Remove Vertex

$O(V+E)$

Add Edge

$O(1)$

Remove Edge

$O(E)$

## Adjacent Matrix

$O(n^2)$

$O(n^2)$

$O(1)$

$O(1)$

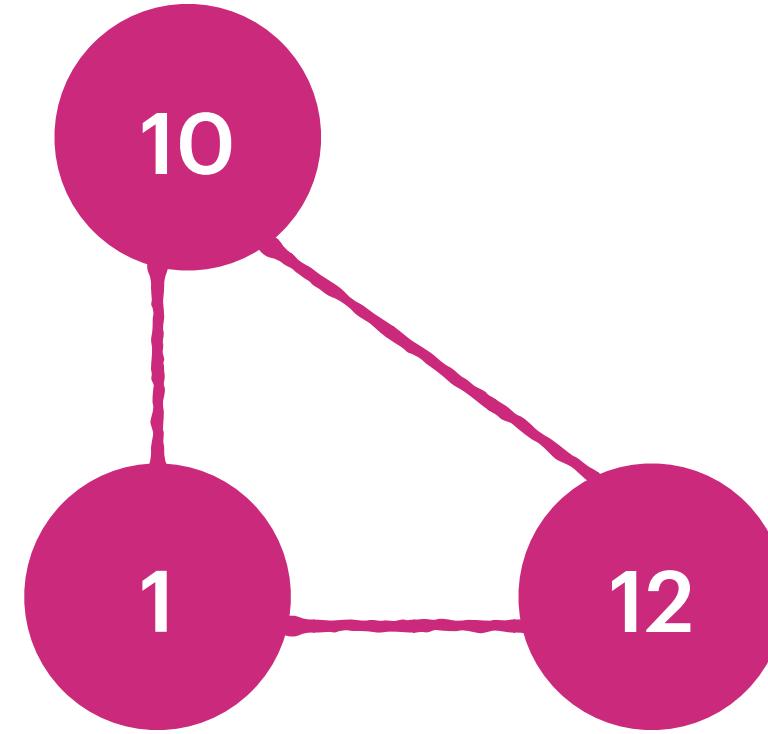
## Design Graph :

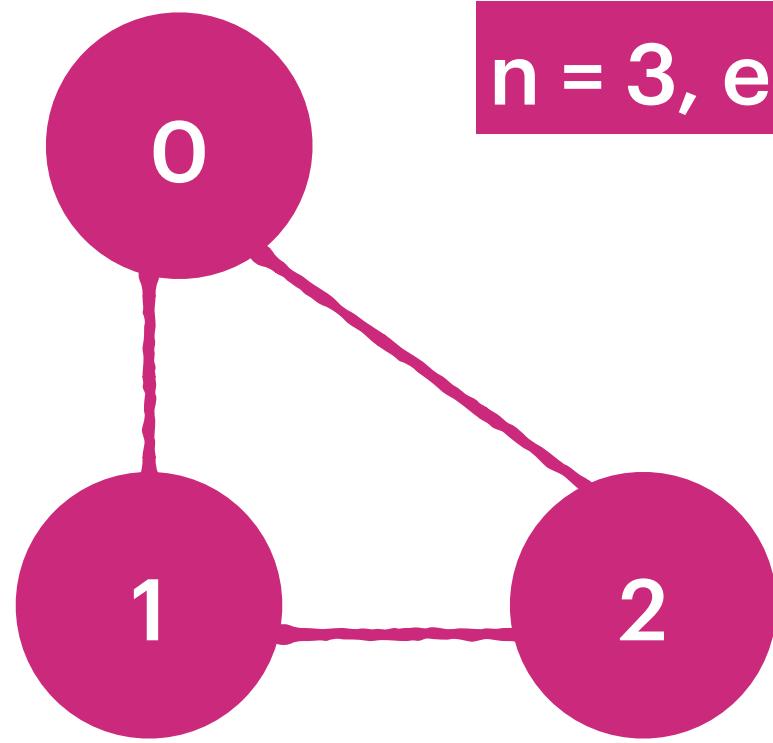
n = 4, edges = [[0,1],[0,2],[1,0],[2,0],[1,3],[2,3][3,1][3,2]]

n = 4, edges = [[0,1],[0,2],[1,3],[2,3]]

n = 6, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]]

System.out.println(unDirectedGraph.dfs());





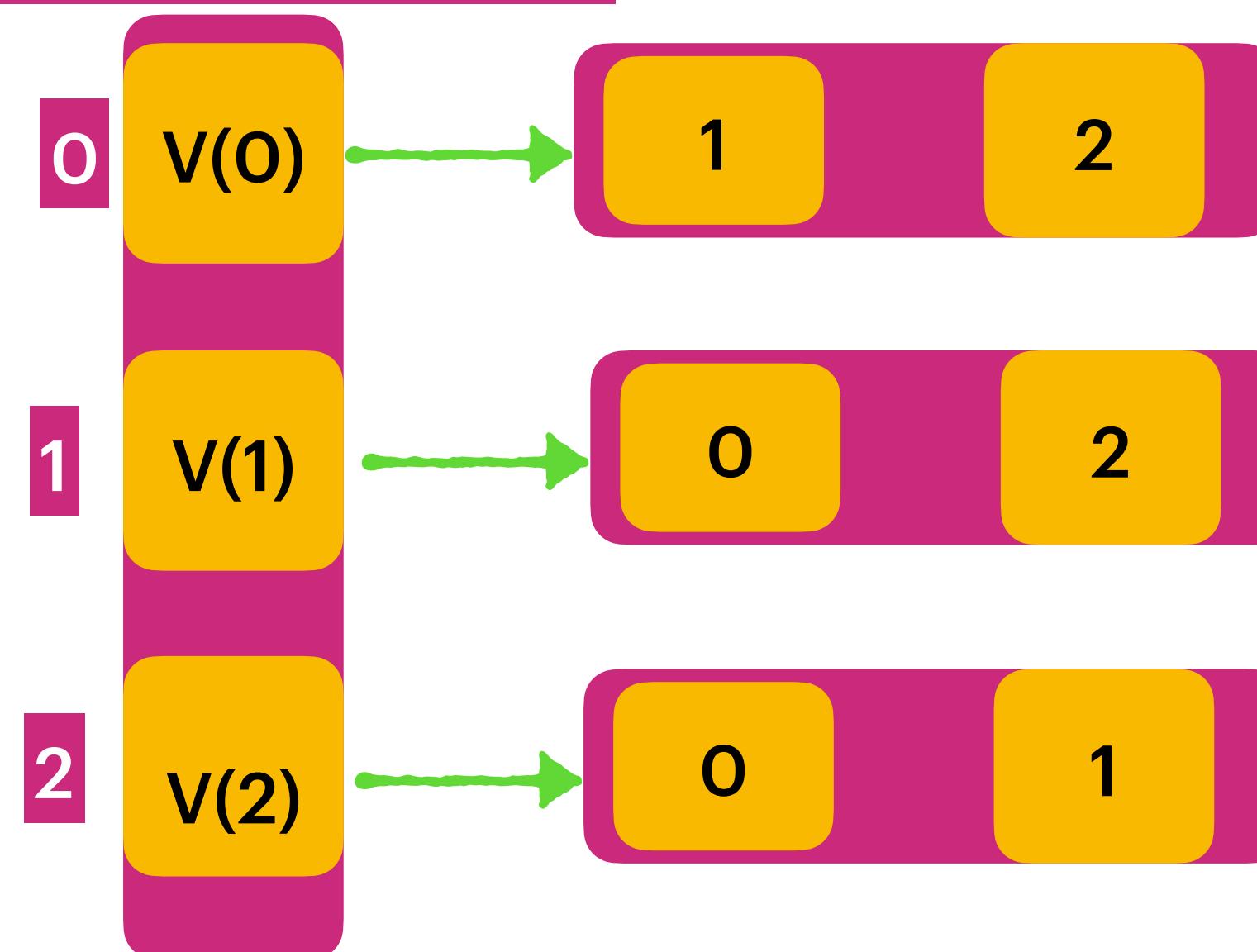
$n = 3$ , edges =  $[[0,1], [0,2], [1,2], [2,1], [2,0], [1,0]]$

List< LinkedHashSet >

Is this can be applied in  
RealWorld ?

Limitations \*\*\*

1. Graph should be fixed .
2. Demands vertexes must  
Be in the range of 0 to n-1
3. Occupy the fixed  
memory.



Hashing

Add Vertex

O(1)

Remove Vertex

O(V)

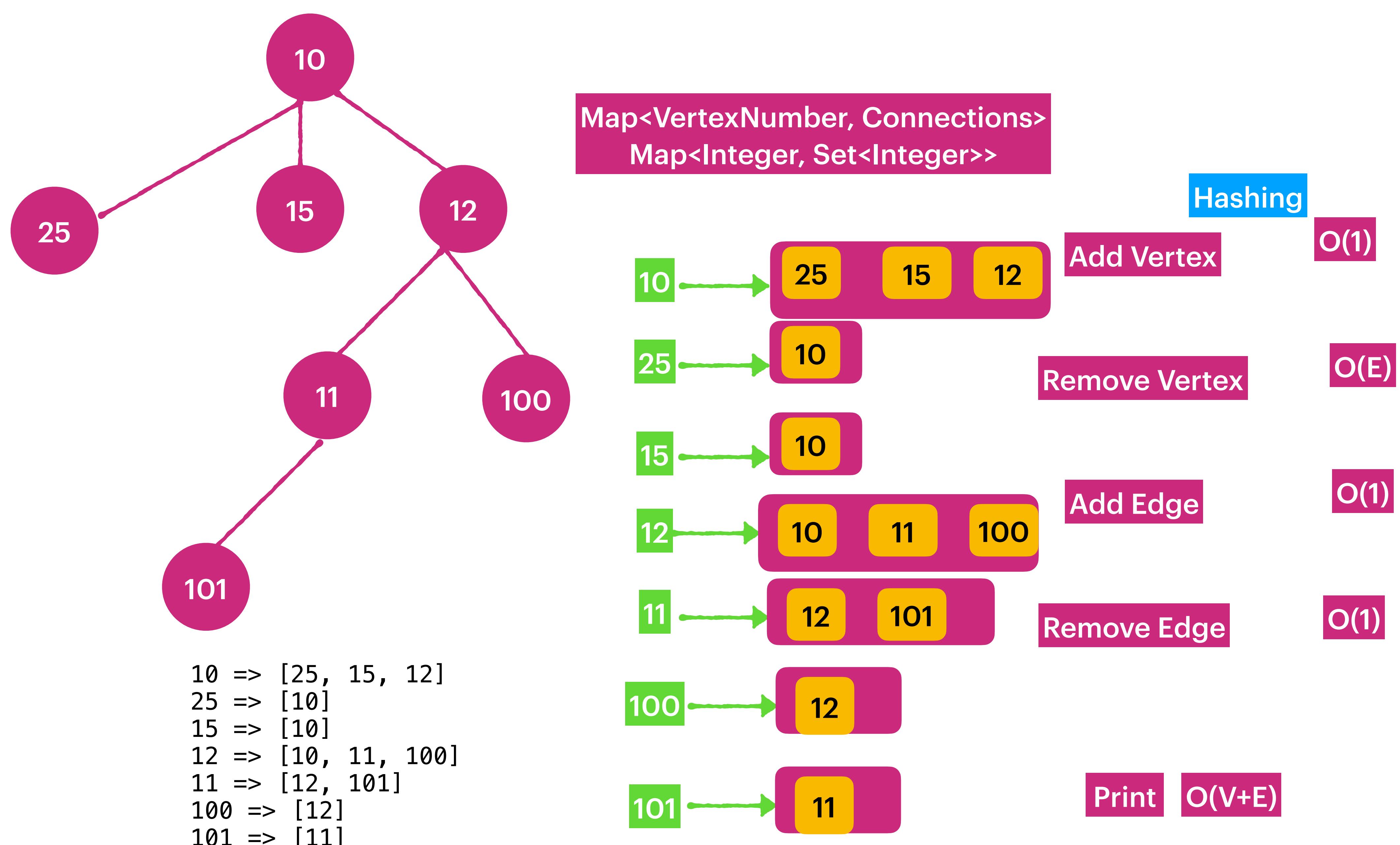
Add Edge

O(1)

Remove Edge

O(1)

Print O(V+E)

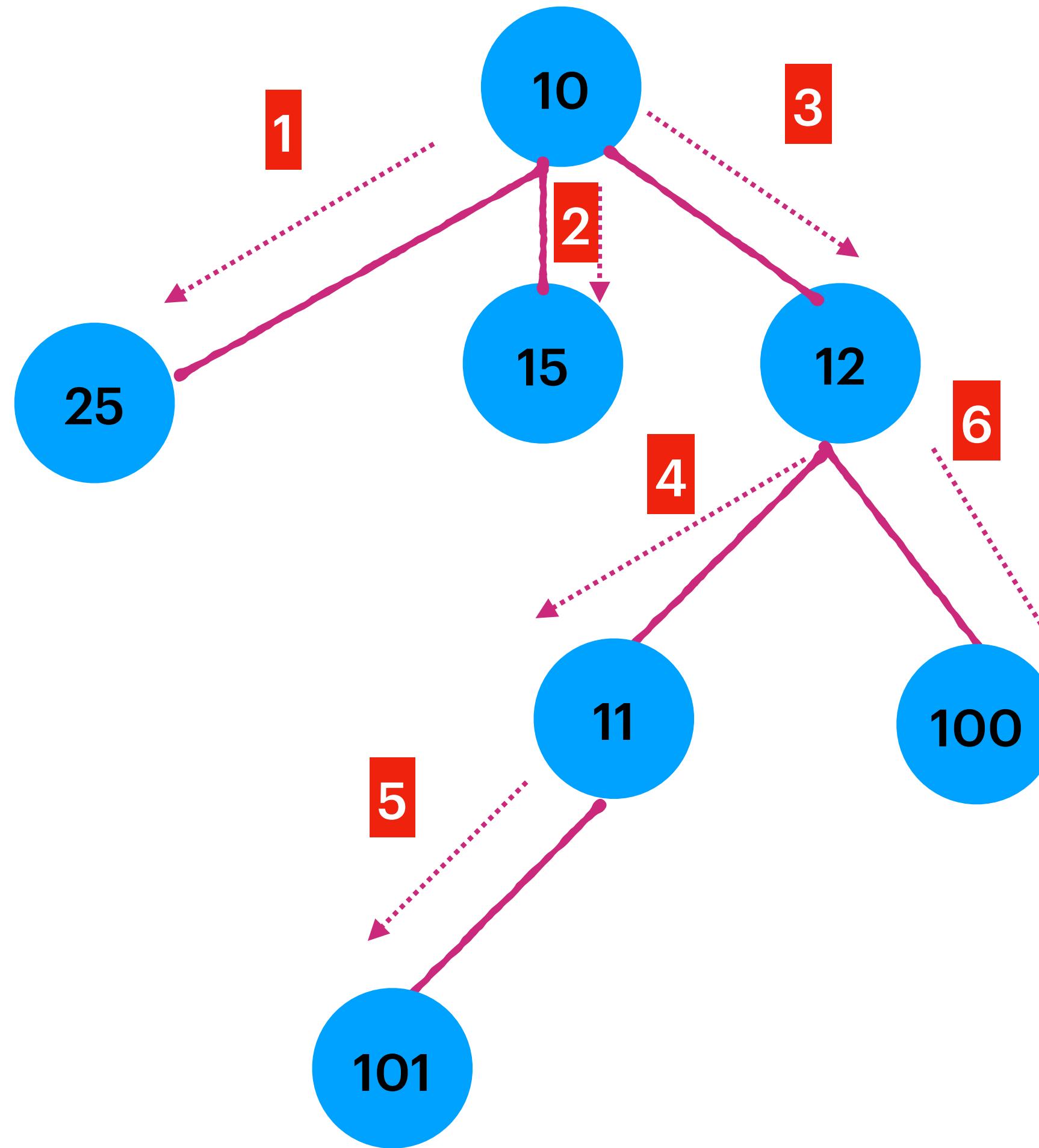


Depth First Search

Undirected Graph

Moving => from root- left to right

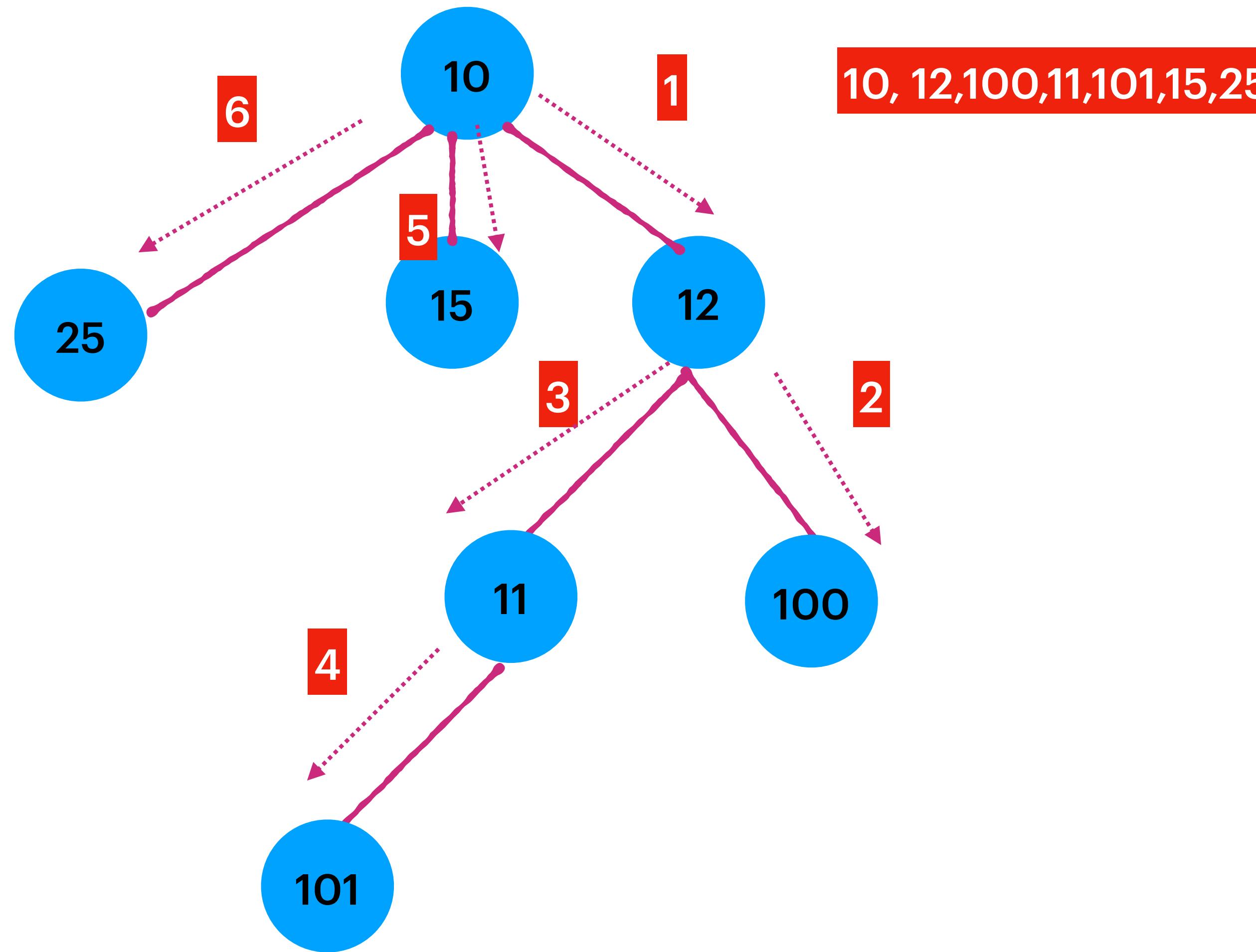
10,25,15,12,11,101,100



## Undirected Graph

### Depth First Search

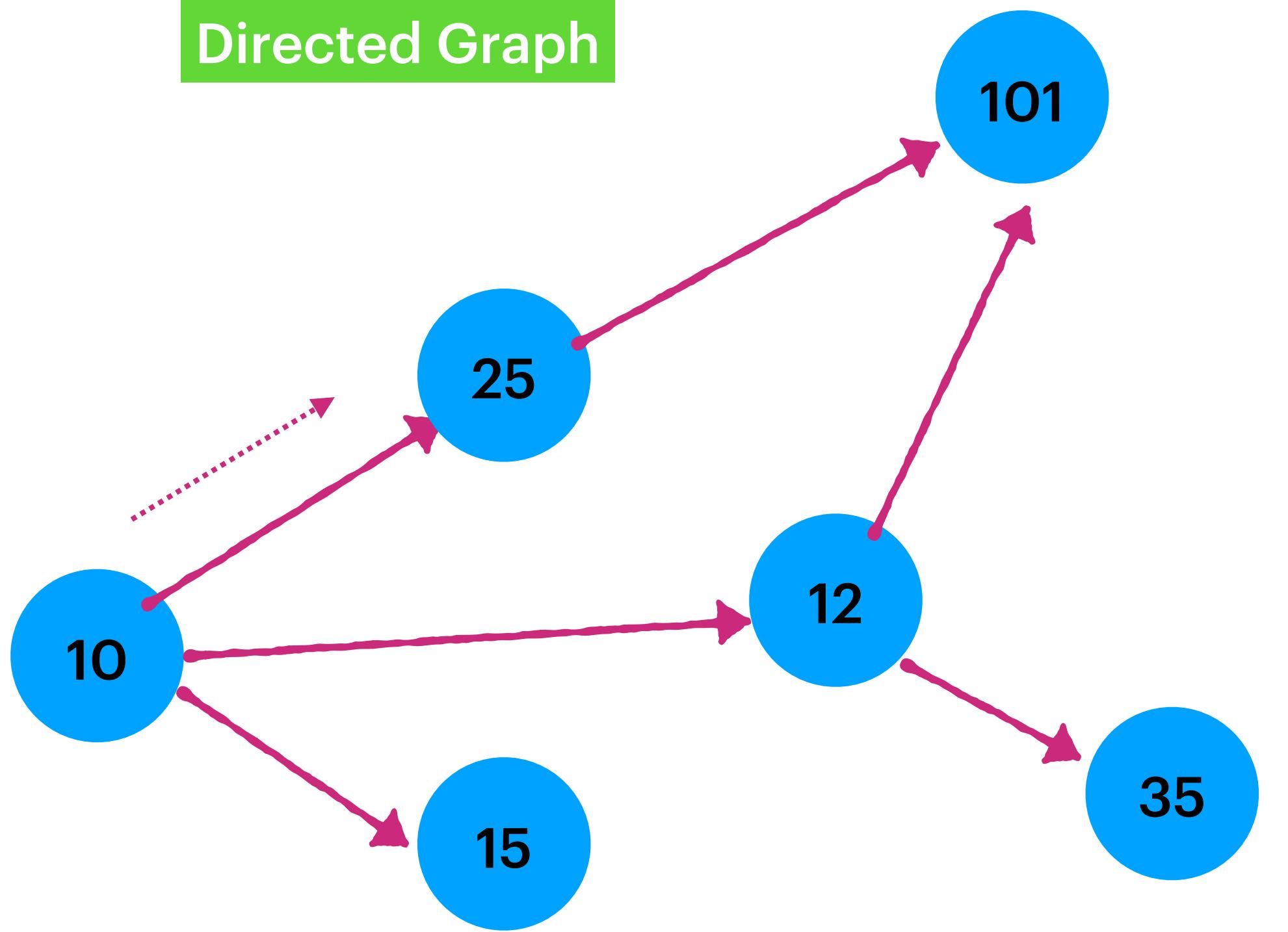
Moving => from root - right to left



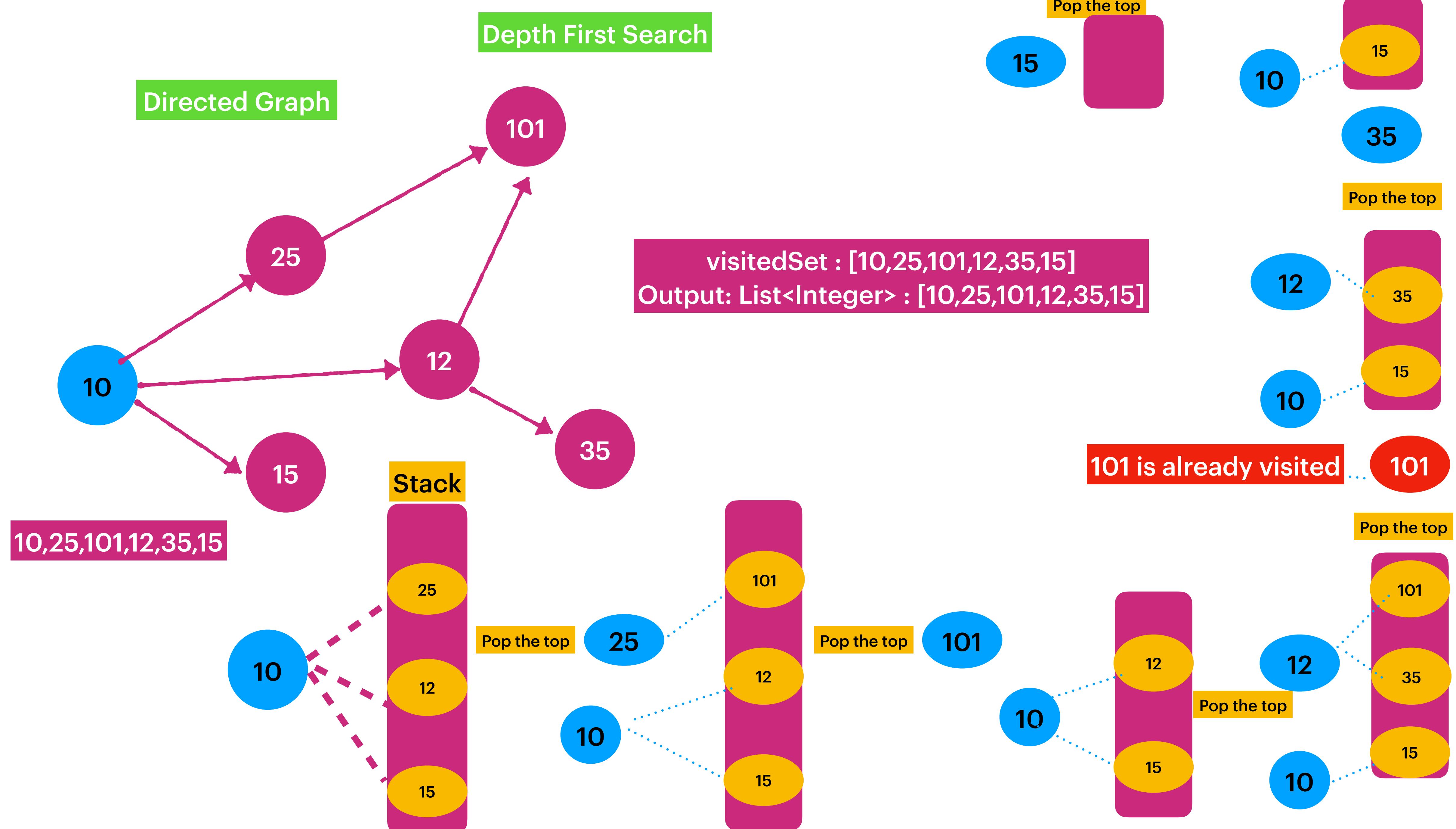
10, 12, 100, 11, 101, 15, 25

## Depth First Search

### Directed Graph

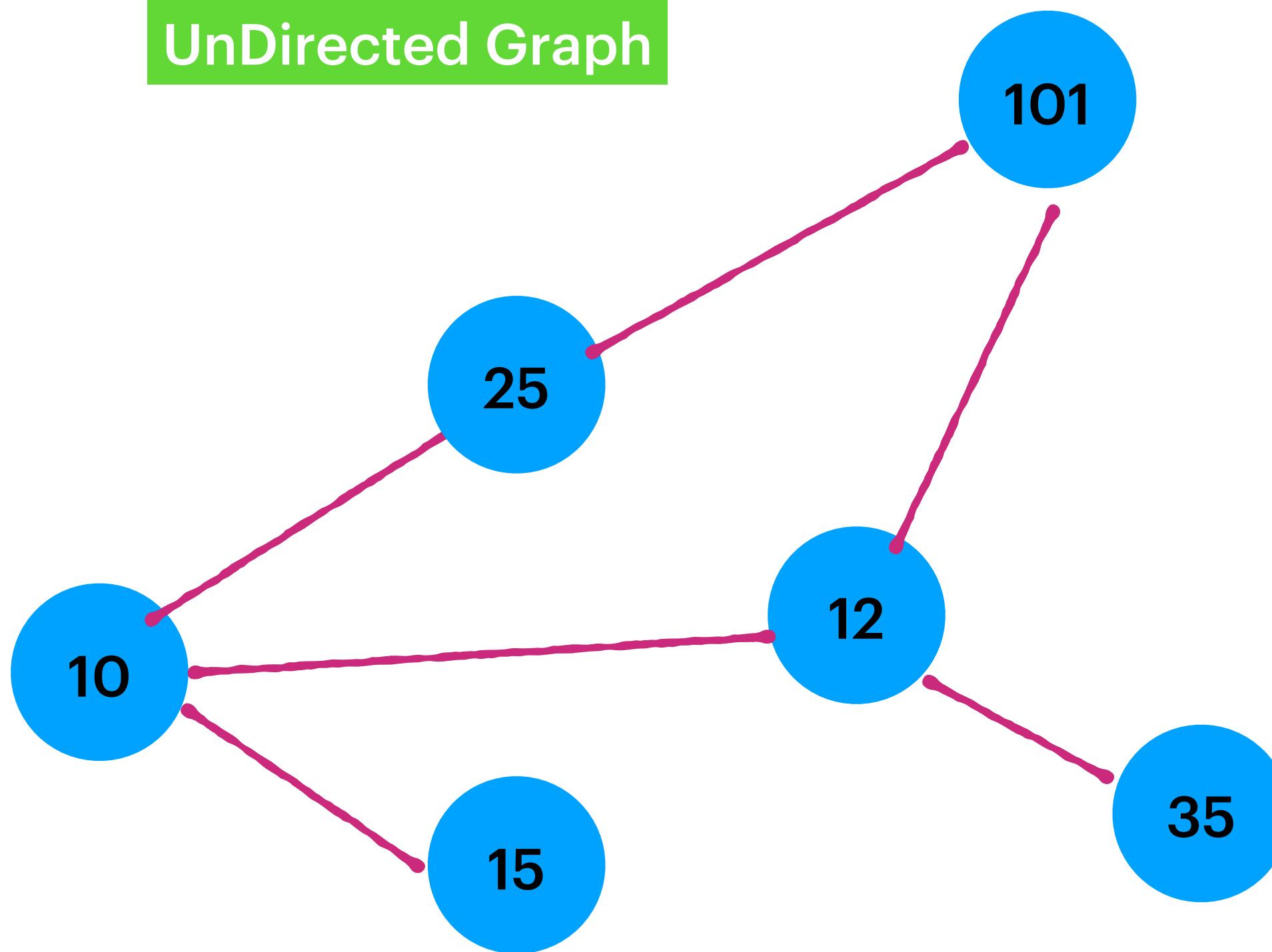


10,25,101,12,35,15



## Depth First Search

### UnDirected Graph



10 => [15, 12, 25] 15 => [10]  
12 => [10, 101, 35]  
25 => [10, 101]  
101 => [25, 12]  
35 => [12]

Stack []  
Set [10,25,101,12,35,15]  
OutputList[10,25,101,12,35,15]

DFS [10, 25, 101, 12, 35, 15]

Traverse All the Vertexes + Visit the unique Edges  
Time Complexity : O(V+E)

$V(10) \Rightarrow [15,12,25]$   
 $V(25) \Rightarrow [10,101]$   
 $V(101) \Rightarrow [25,12]$   
 $V(12) \Rightarrow [10,101,35]$   
 $V(35) \Rightarrow [12]$   
 $V(15) \Rightarrow [10]$

start : 10  
stack( $v(10)$ ) : 1

$v(10) \Rightarrow$  stack ([15,12,25]) = 3 : visited : [ $v(10)$ ]  
 $v(25) \Rightarrow$  stack ([15,12,10,101]) = 2: visited : [ $v(10), v(25)$ ]  
 $v(101) \Rightarrow$  stack ([15,12,10,25,12]) = 2 visited : [ $v(10), v(25), v(101)$ ]  
 $v(12) \Rightarrow$  stack ([15,12,10,25,10,101,35]) = 3  
visited : [ $v(10), v(25), v(101), v(12)$ ]  
 $V(35) \Rightarrow$  stack ([15,12,10,25,10,101,12]) = 1  
visited : [ $v(10), v(25), v(101), v(12), v(35)$ ]

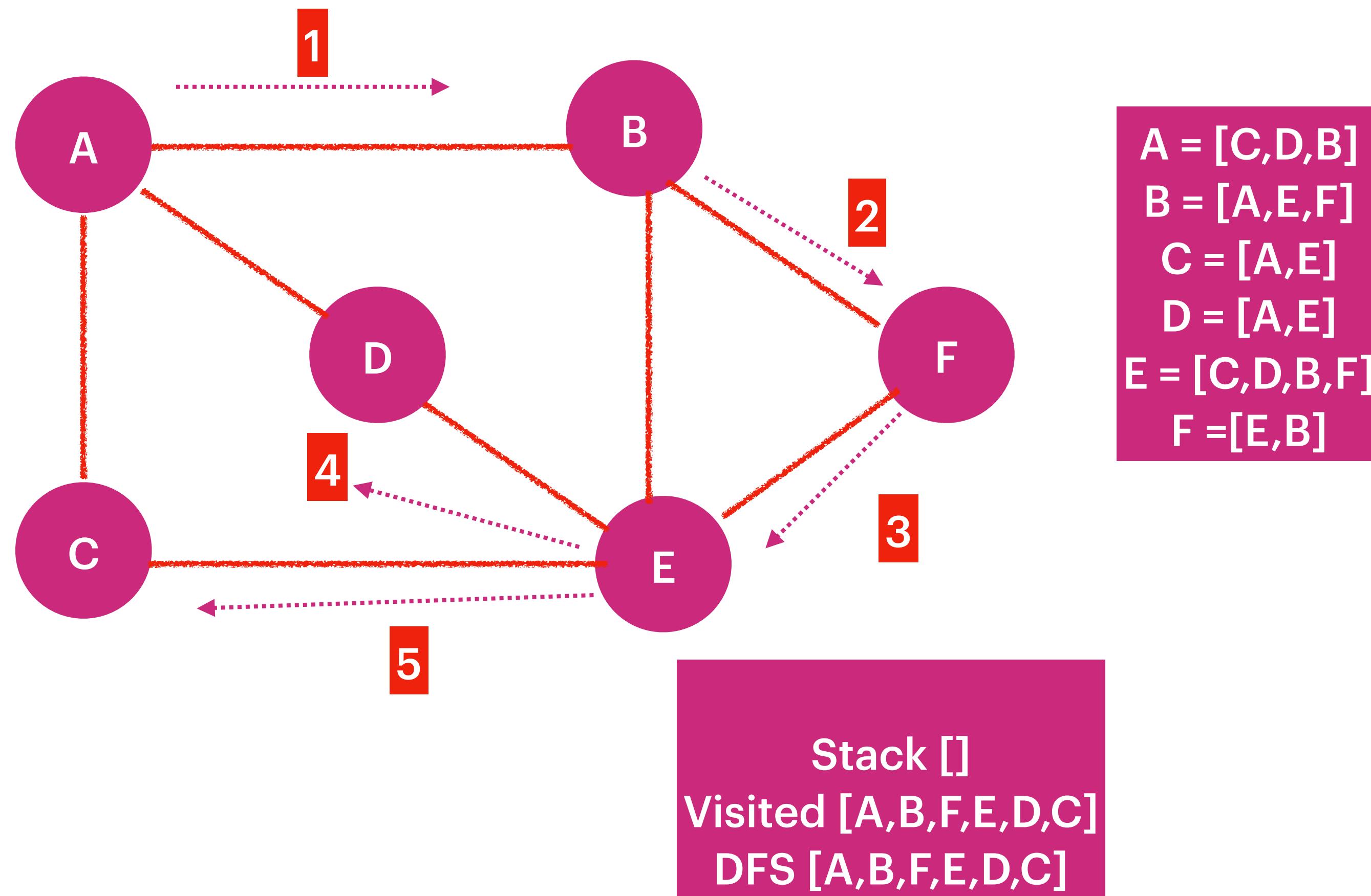
$v(12) = 0$  stack ([15,12,10,25,10,101])  
 $v(101) = 0$  stack ([15,12,10,25,10])  
 $v(10) = 0$  stack ([15,12,10,25])  
 $v(25) = 0$  stack ([15,12,10])  
 $v(10) = 0$  stack ([15,12])  
 $v(12) = 0$  stack ([15])  
 $v(15) = 1$  stack ([10]) visited : [ $v(10), v(25), v(101), v(12), v(35), v(15)$ ]  
 $v(10) = Ostac([])$  visited : [ $v(10), v(25), v(101), v(12), v(35), v(15)$ ]

numberOf.Connections(V) + Unique[Edges]

$$13V + 12(E) = V + E$$

Analysis's Of DFS Time Complexity :  $O(V+E)$

## Depth First Search

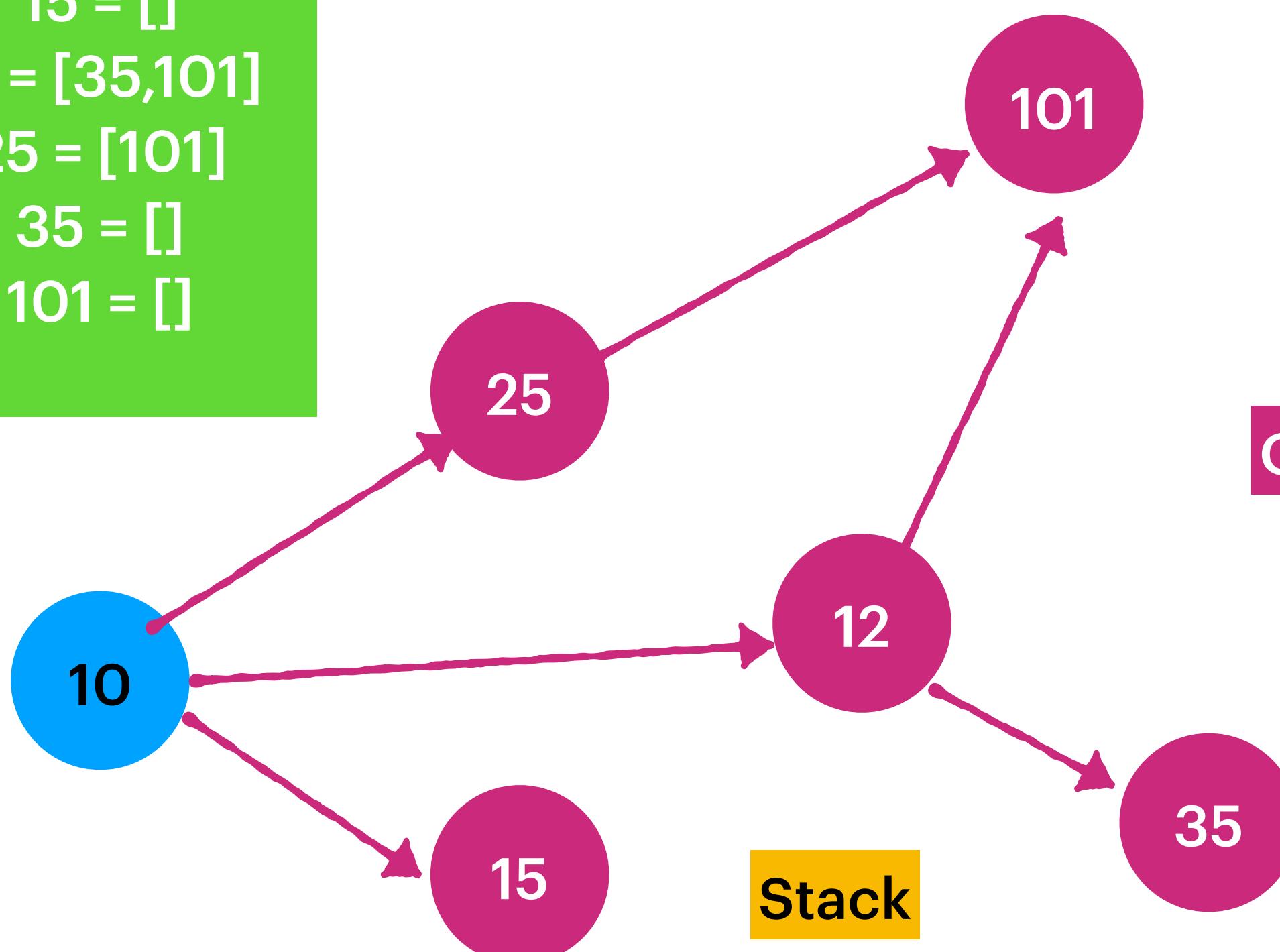


**Directed Graph**

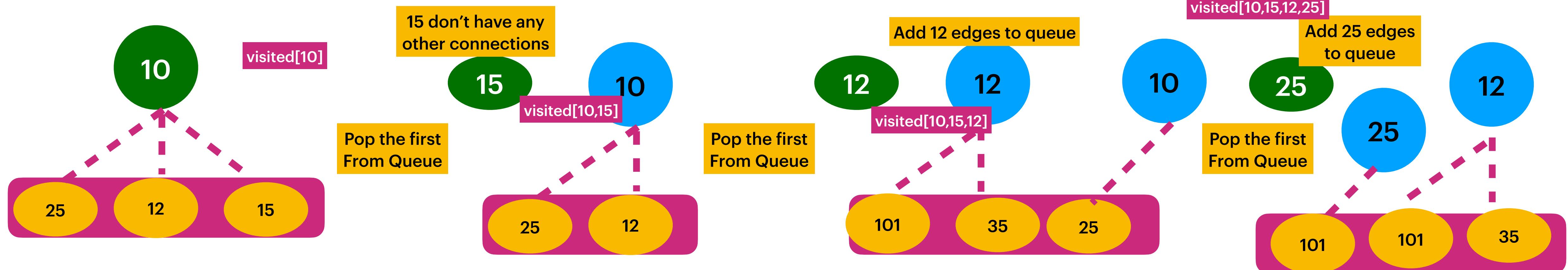
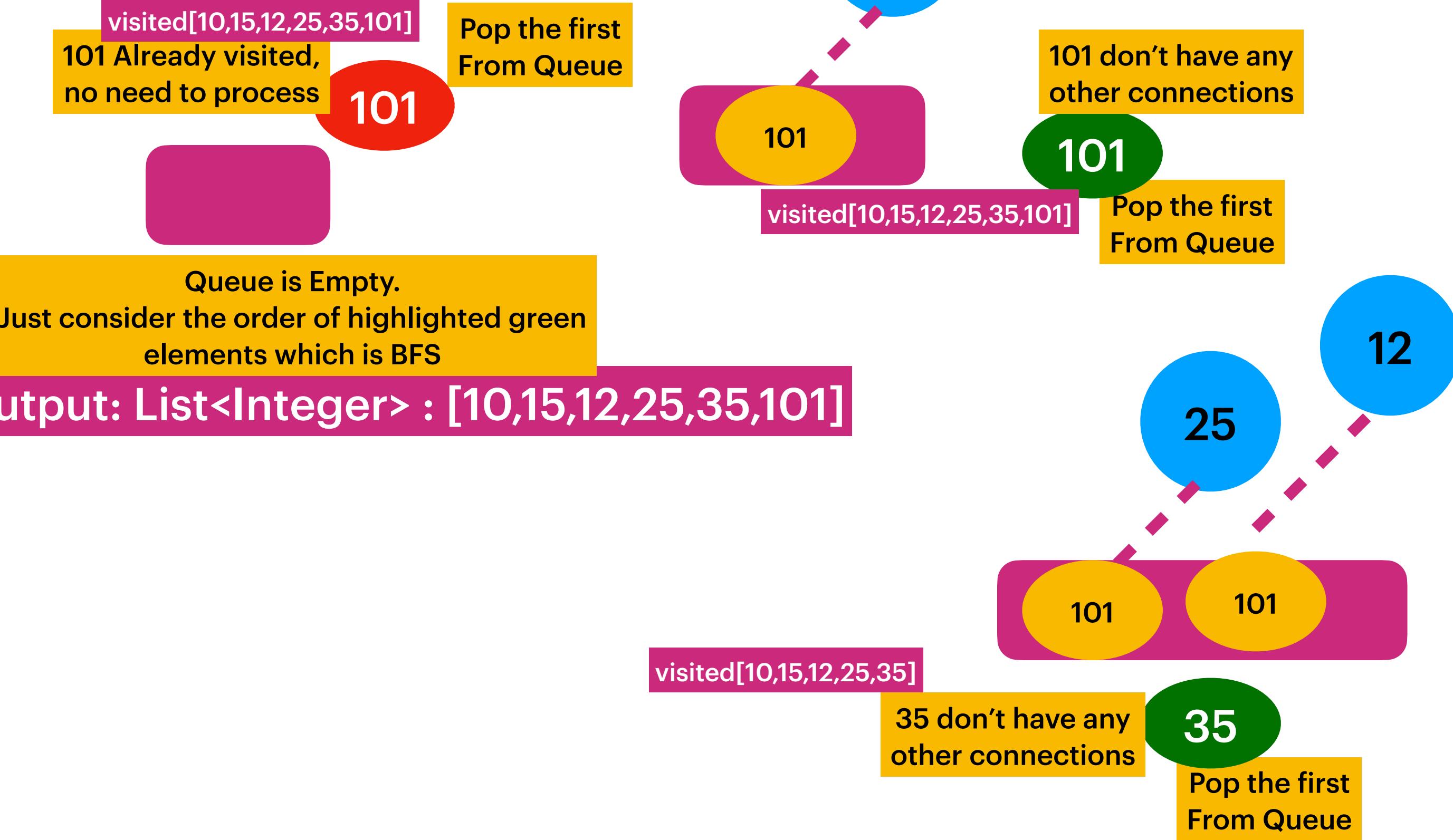
```

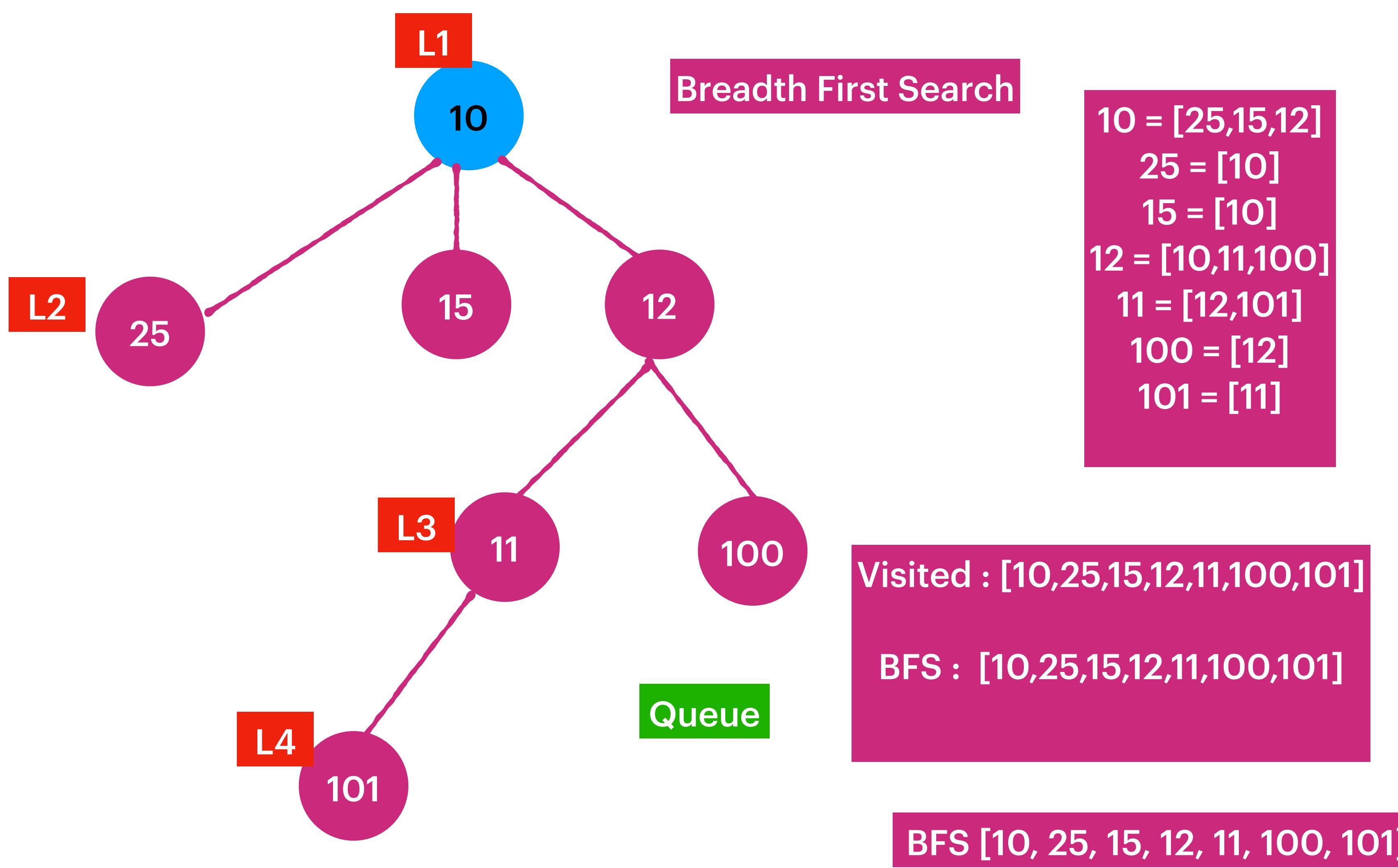
10 = [15,12,25]
15 = []
12 = [35,101]
25 = [101]
35 = []
101 = []

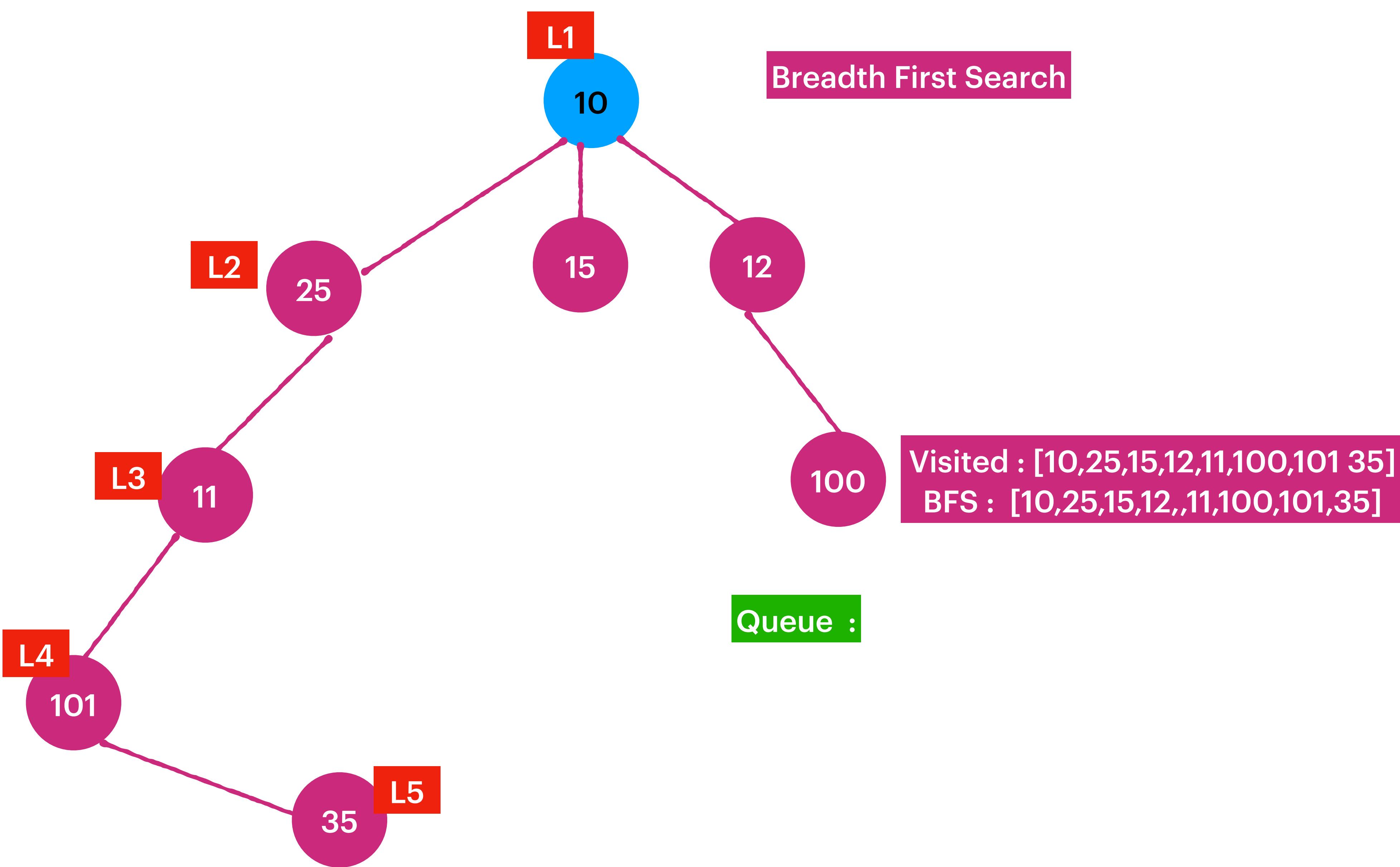
```



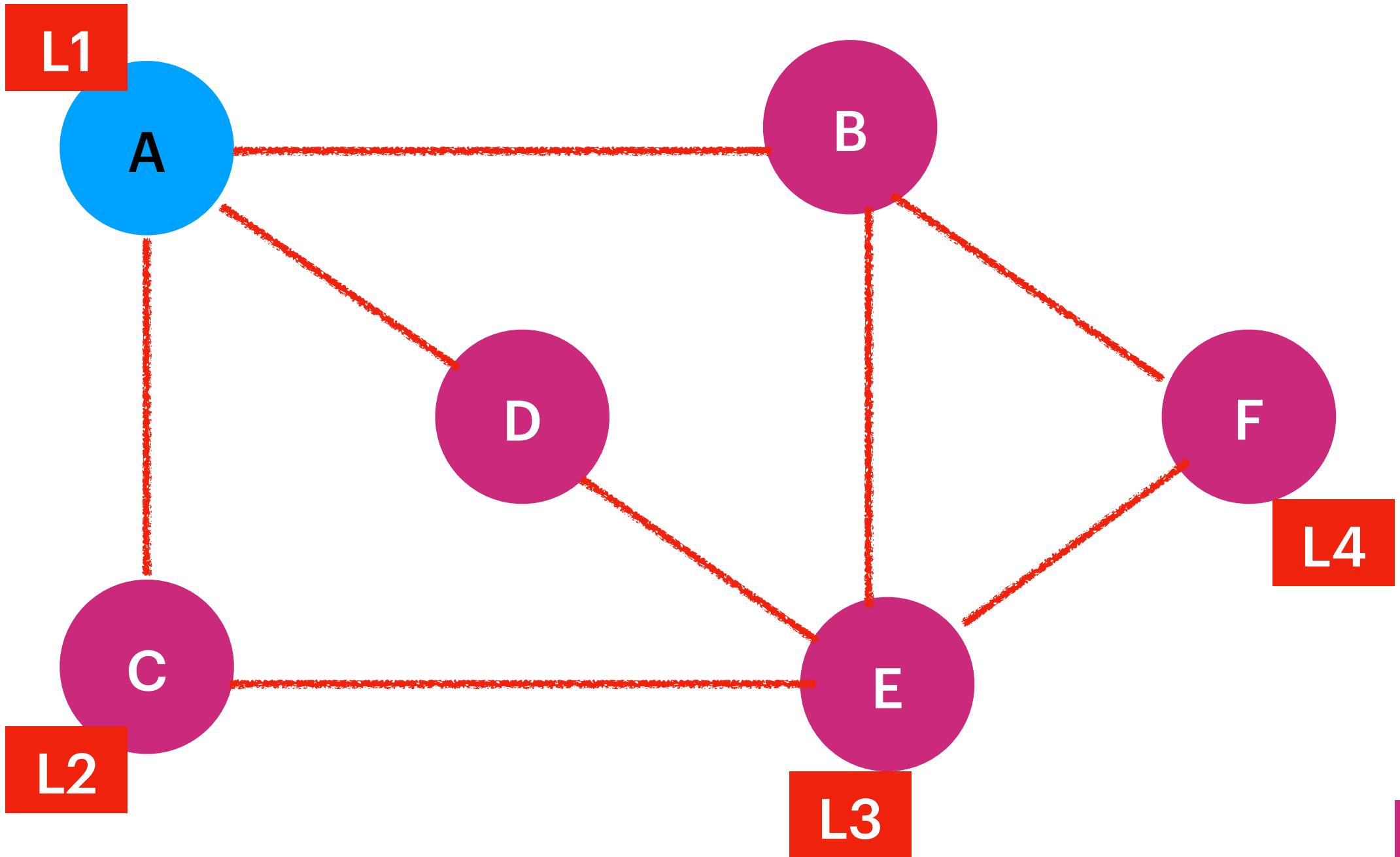
## Breadth First Search







## Breadth First Search



A = [C,D,B]  
B = [A,E,F]  
C = [A,E]  
D = [A,E]  
E = [C,D,B,F]  
F = [E,B]

BFS : A,C,D,B,E,F

## Find if Path Exists in Graph

There is a bi-directional graph with  $n$  vertices, where each vertex is labeled from 0 to  $n - 1$  (inclusive). The edges in the graph are represented as a 2D integer array  $\text{edges}$ , where each  $\text{edges}[i] = [\text{ui}, \text{vi}]$  denotes a bi-directional edge between vertex  $\text{ui}$  and vertex  $\text{vi}$ . Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.

You want to determine if there is a valid path that exists from vertex  $\text{start}$  to vertex  $\text{end}$ .

Given  $\text{edges}$  and the integers  $n$ ,  $\text{start}$ , and  $\text{end}$ , return true if there is a valid path from  $\text{start}$  to  $\text{end}$ , or false otherwise.

**Input:**  $n = 3$ ,  $\text{edges} = [[0,1],[1,2],[2,0]]$ ,  $\text{start} = 0$ ,  $\text{end} = 2$

**Output:** true

**Explanation:** There are two paths from vertex 0 to vertex 2:

- $0 \rightarrow 1 \rightarrow 2$
- $0 \rightarrow 2$

**Constraints:**

$1 \leq n \leq 2 * 10^5$

$0 \leq \text{edges.length} \leq 2 * 10^5$

$\text{edges}[i].length == 2$

$0 \leq u_i, v_i \leq n - 1$

$u_i \neq v_i$

$0 \leq \text{start}, \text{end} \leq n - 1$

There are no duplicate edges.

There are no self edges.

**Input:**  $n = 6$ ,  $\text{edges} = [[0,1],[0,2],[3,5],[5,4],[4,3]]$ ,  $\text{start} = 0$ ,  $\text{end} = 5$

**Output:** false

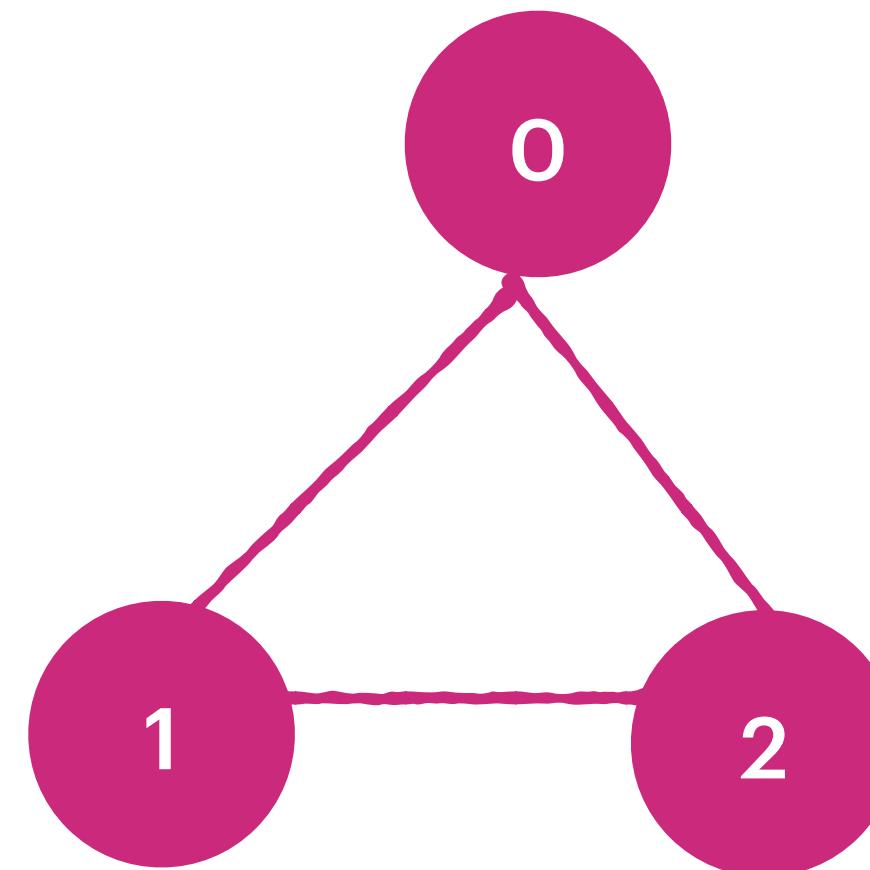
**Explanation:** There is no path from vertex 0 to vertex 5.

**Input:** n = 3, edges = [[0,1],[1,2],[2,0]], start = 0, end = 2

**Output:** true

**Explanation:** There are two paths from vertex 0 to vertex 2:

- 0 → 1 → 2
- 0 → 2



**Solution 1 DisJoint Set :** O(1) :

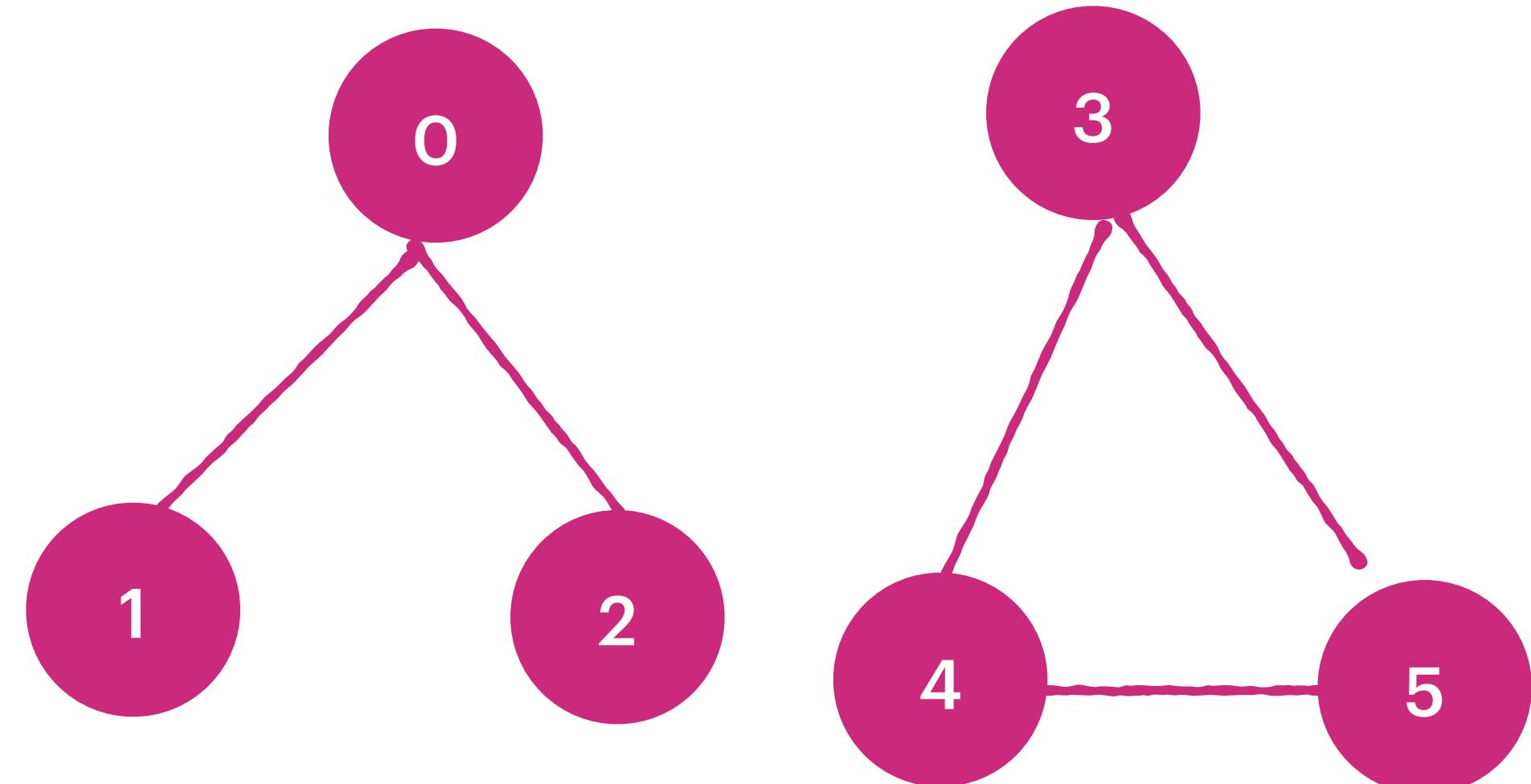
**Solution 2. DFS :** O(V+E)

**Solution 3 BFS :** O(V+E)

**Input:** n = 6, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]], start = 0, end = 5

**Output:** false

**Explanation:** There is no path from vertex 0 to vertex 5.



## All Paths From Source to Target

Given a directed acyclic graph (DAG) of  $n$  nodes labeled from 0 to  $n - 1$ , find all possible paths from node 0 to node  $n - 1$  and return them in any order. The graph is given as follows:  $\text{graph}[i]$  is a list of all nodes you can visit from node  $i$  (i.e., there is a directed edge from node  $i$  to node  $\text{graph}[i][j]$ ).

**Input:**  $\text{graph} = [[1,2],[3],[3],[]]$

**Output:**  $[[0,1,3],[0,2,3]]$

**Explanation:** There are two paths:  $0 \rightarrow 1 \rightarrow 3$  and  $0 \rightarrow 2 \rightarrow 3$ .

**Input:**  $\text{graph} = [[4,3,1],[3,2,4],[3],[4],[]]$

**Output:**  $[[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]$

$n == \text{graph.length}$

$2 \leq n \leq 15$

$0 \leq \text{graph}[i][j] < n$

$\text{graph}[i][j] \neq i$  (i.e., there will be no self-loops).

All the elements of  $\text{graph}[i]$  are unique.

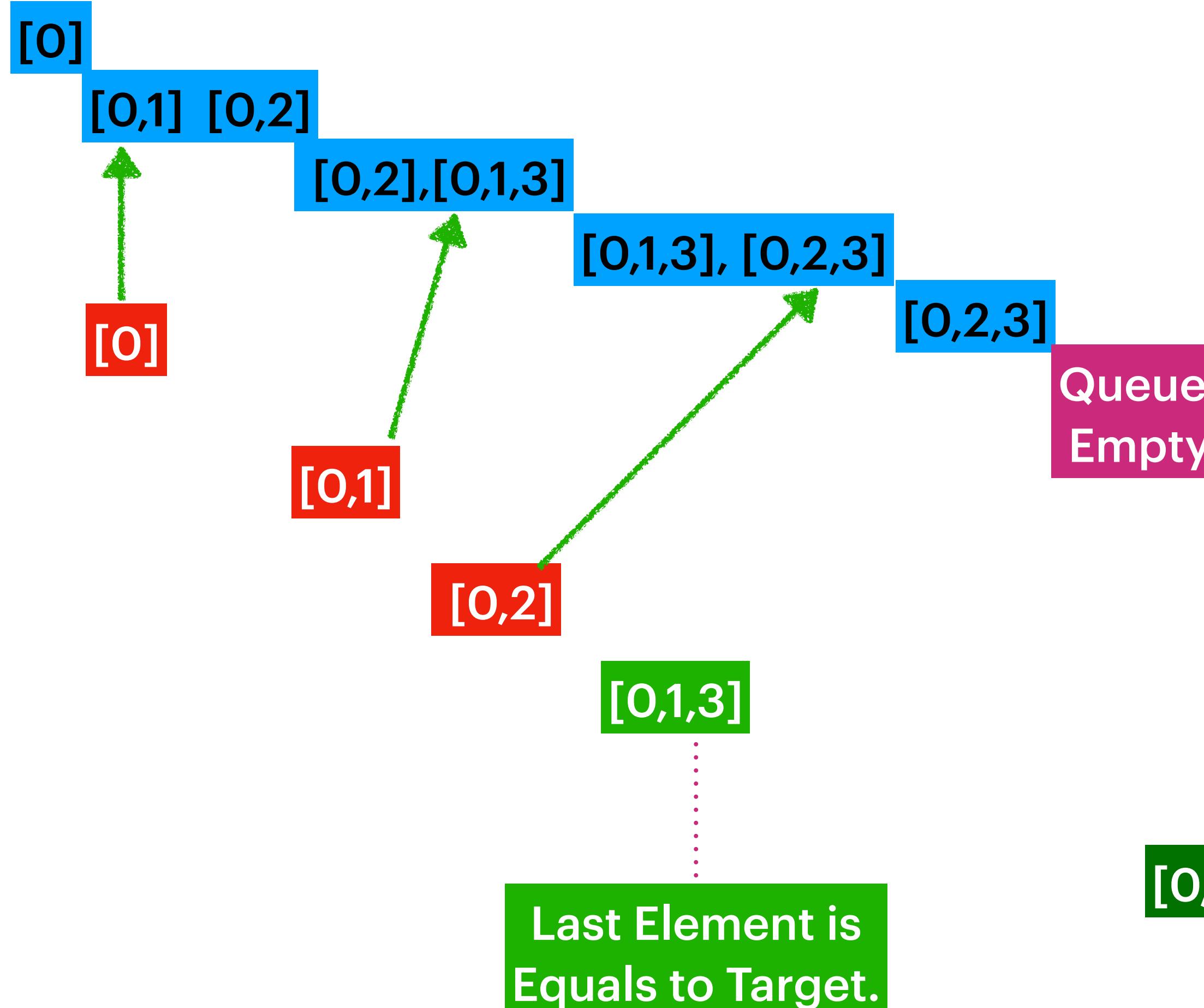
The input graph is guaranteed to be a DAG.

## All Paths From Source to Target

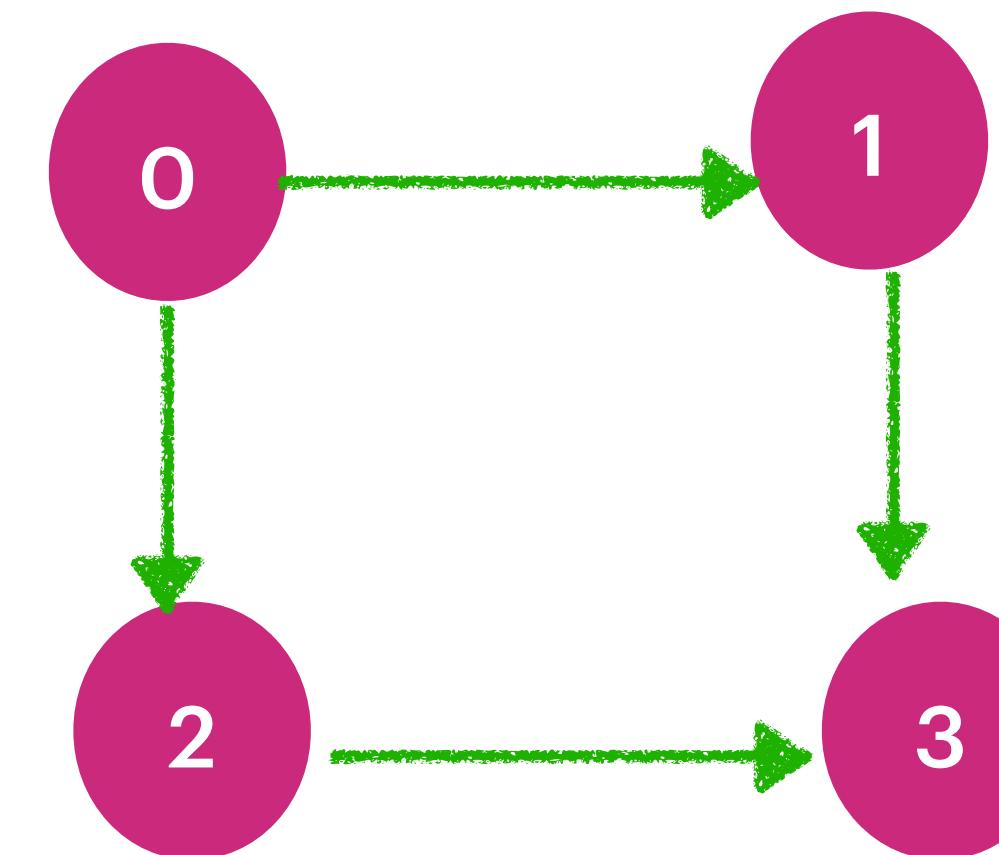
Input: graph = [[1,2],[3],[3],[]] n=4

Output: [[0,1,3],[0,2,3]]

Explanation: There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.



graph = [ [1,2], [3], [3], [] ] n=4



Paths From 0 to 3

0 → 1 → 3 , 0 → 2 → 3

[0,2,3]

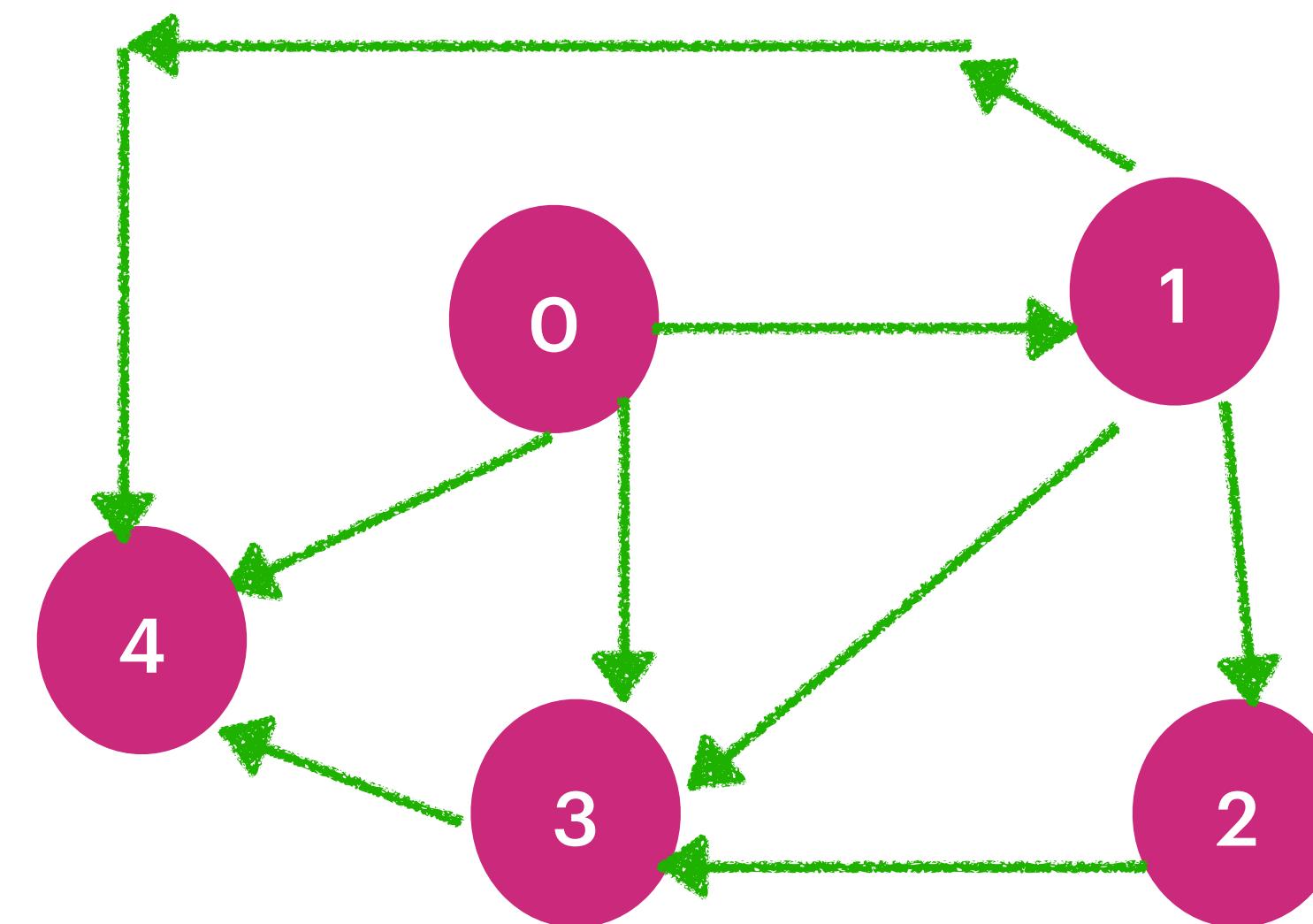
Last Element is Equals to Target.

## All Paths From Source to Target

`n == graph.length`  
`2 <= n <= 15`  
`0 <= graph[i][j] < n`  
`graph[i][j] != i` (i.e., there will be no self-loops).  
All the elements of `graph[i]` are unique.  
The input graph is guaranteed to be a DAG.

`graph = [ [4,3,1], [3,2,4], [3], [4], [] ] n=5`

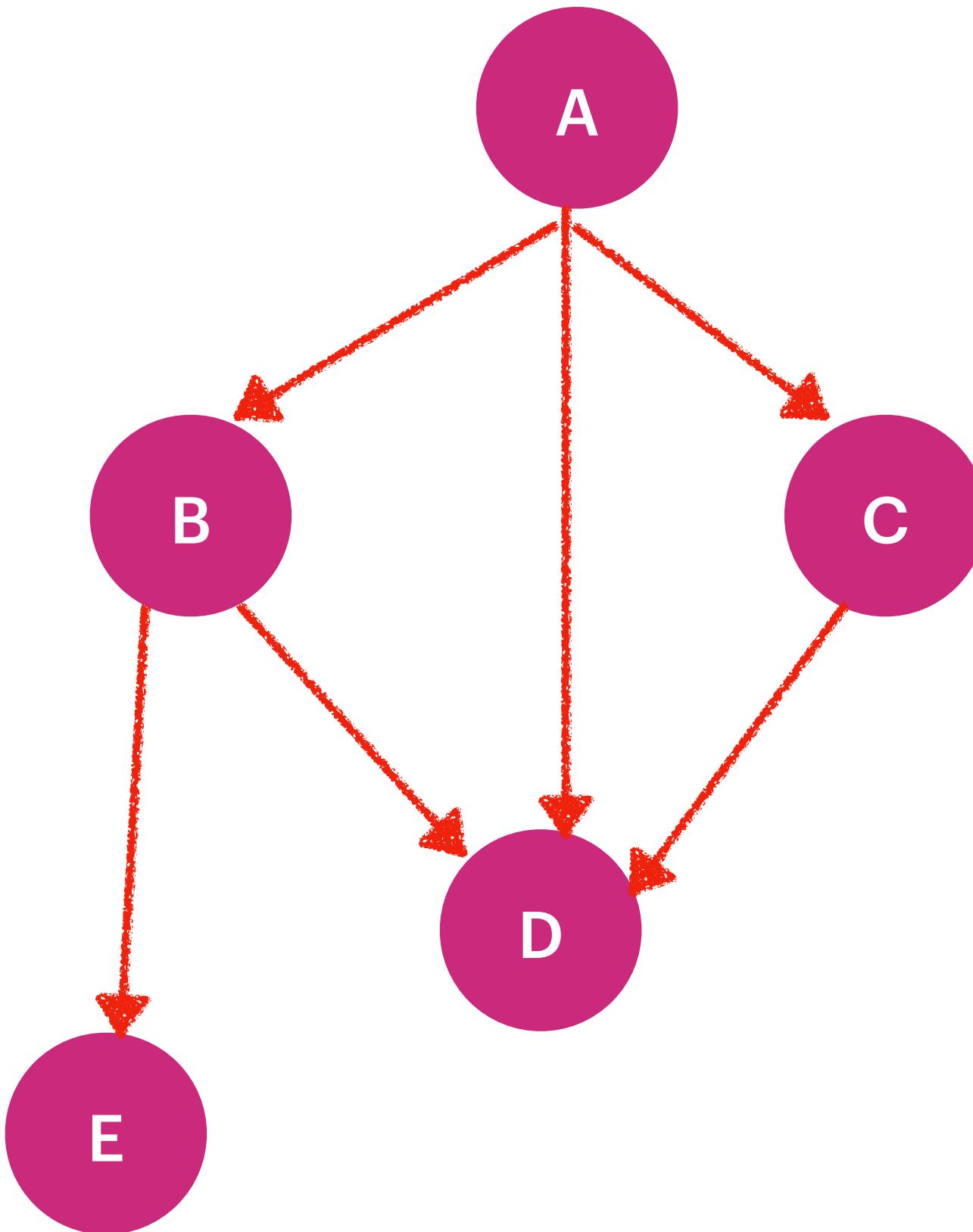
**Input:** `graph = [[4,3,1],[3,2,4],[3],[4],[]] n=5`  
**Output:** `[[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]`



## Paths From 0 to 4

`0->4 , 0->3->4, 0->1->3->4, 0->1->2->3->4, 0->1->4`

BFS  
Source : A  
Target : D



[A]

[A,B]

[A,C]

[A,D]

Last Element is  
Equals to Target.

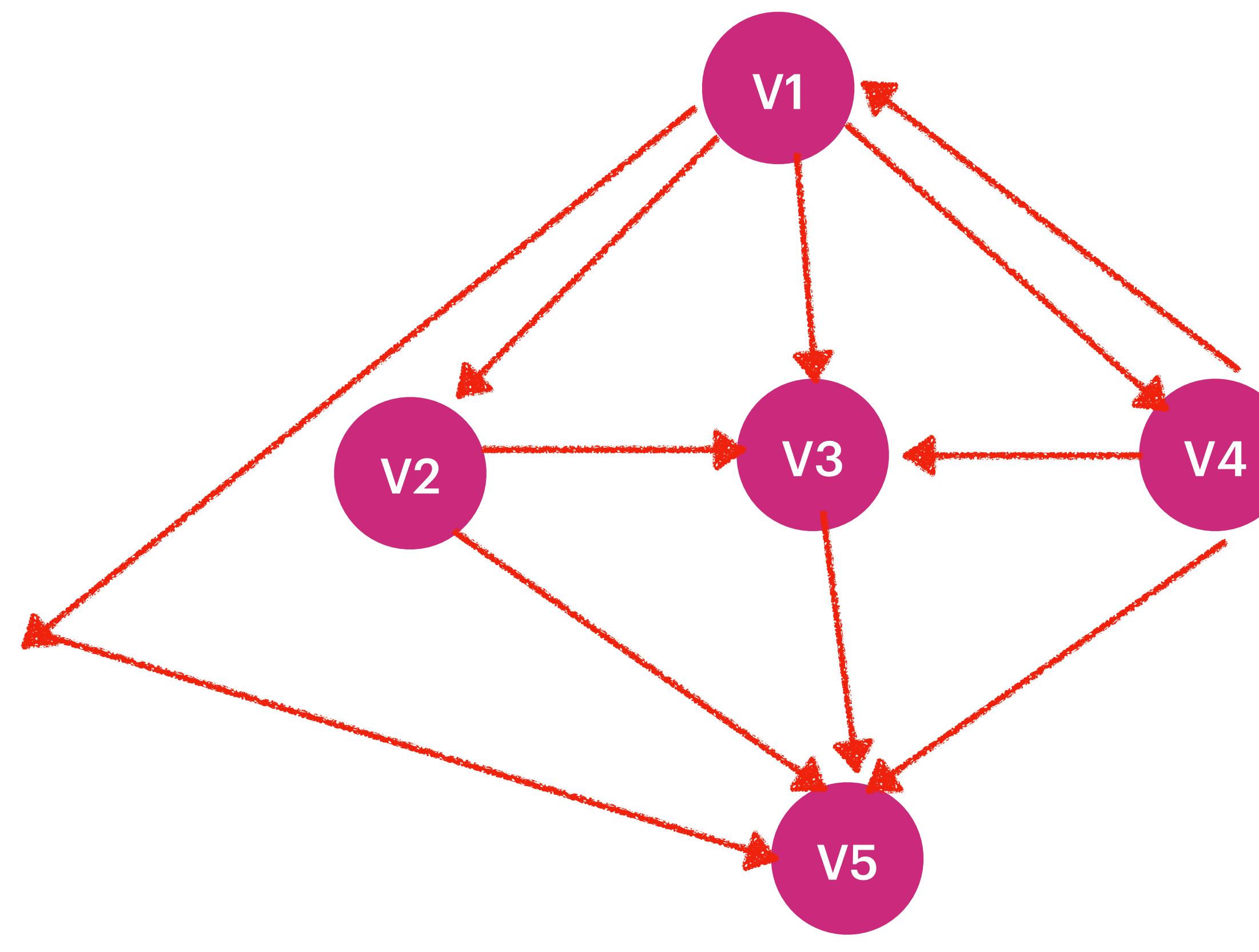
[A,B,D]

Last Element is  
Equals to Target.

[A,B,E]

[A,C,D]

Last Element is  
Equals to Target.



Source  
(V1)  
Target  
(V5)

$V1 \rightarrow V5$  through  $V3$  IFF  $V1 \rightarrow V3$  &  $V3 \rightarrow V5$

$V1 \rightarrow V5$  through  $V2$  IFF  $V1 \rightarrow V2$  &  $V2 \rightarrow V5$

$$V * 2^V$$

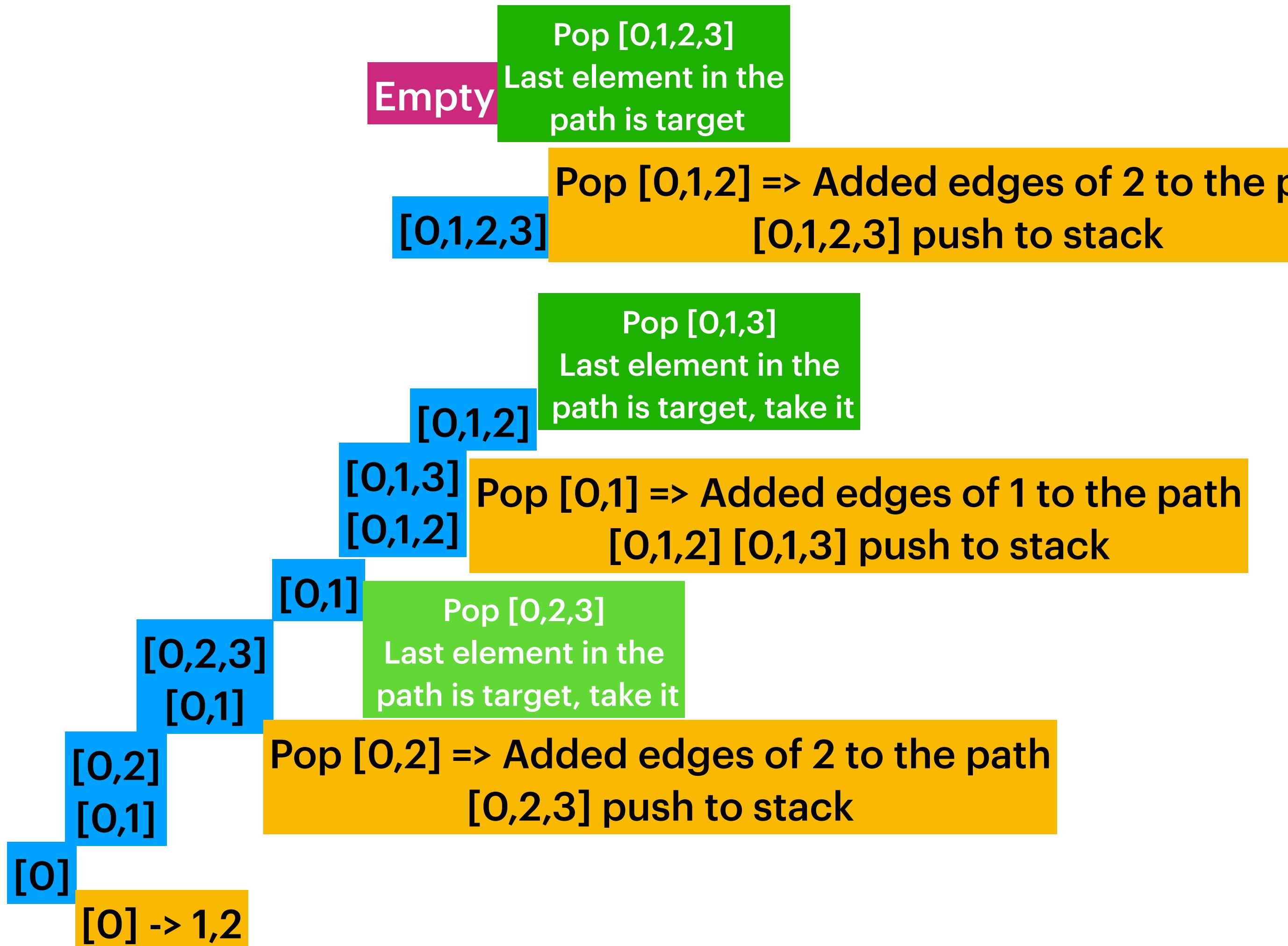
In worst case as per Graph Theory For each vertex we can make  $(2^{V-1} - 1)$  unique paths. So in worst cast time complexity is  $O(V * 2^V)$

## All Paths From Source to Target : DFS

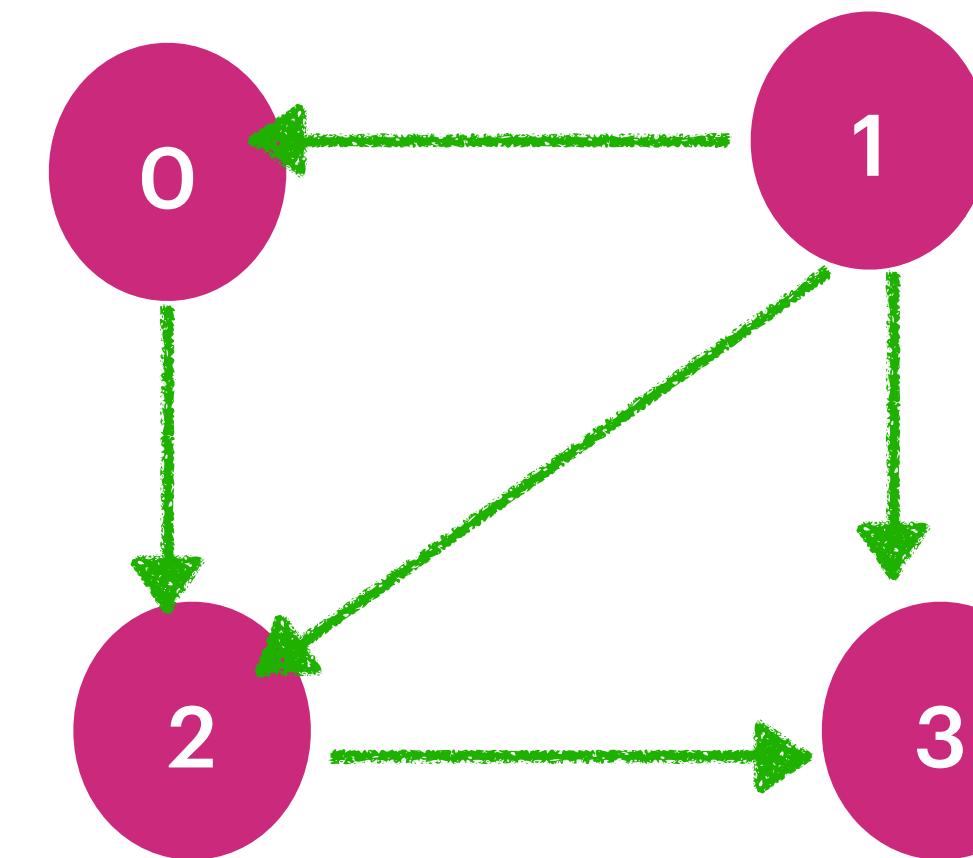
Input: graph = [[1,2],[2,3],[3],[]] n=4

Output: [[0,1,3],[0,2,3],[0,1,2,3]]

Explanation: There are two paths: 0  $\rightarrow$  1  $\rightarrow$  3 and 0  $\rightarrow$  2  $\rightarrow$  3.  
0  $\rightarrow$  1  $\rightarrow$  2  $\rightarrow$  3

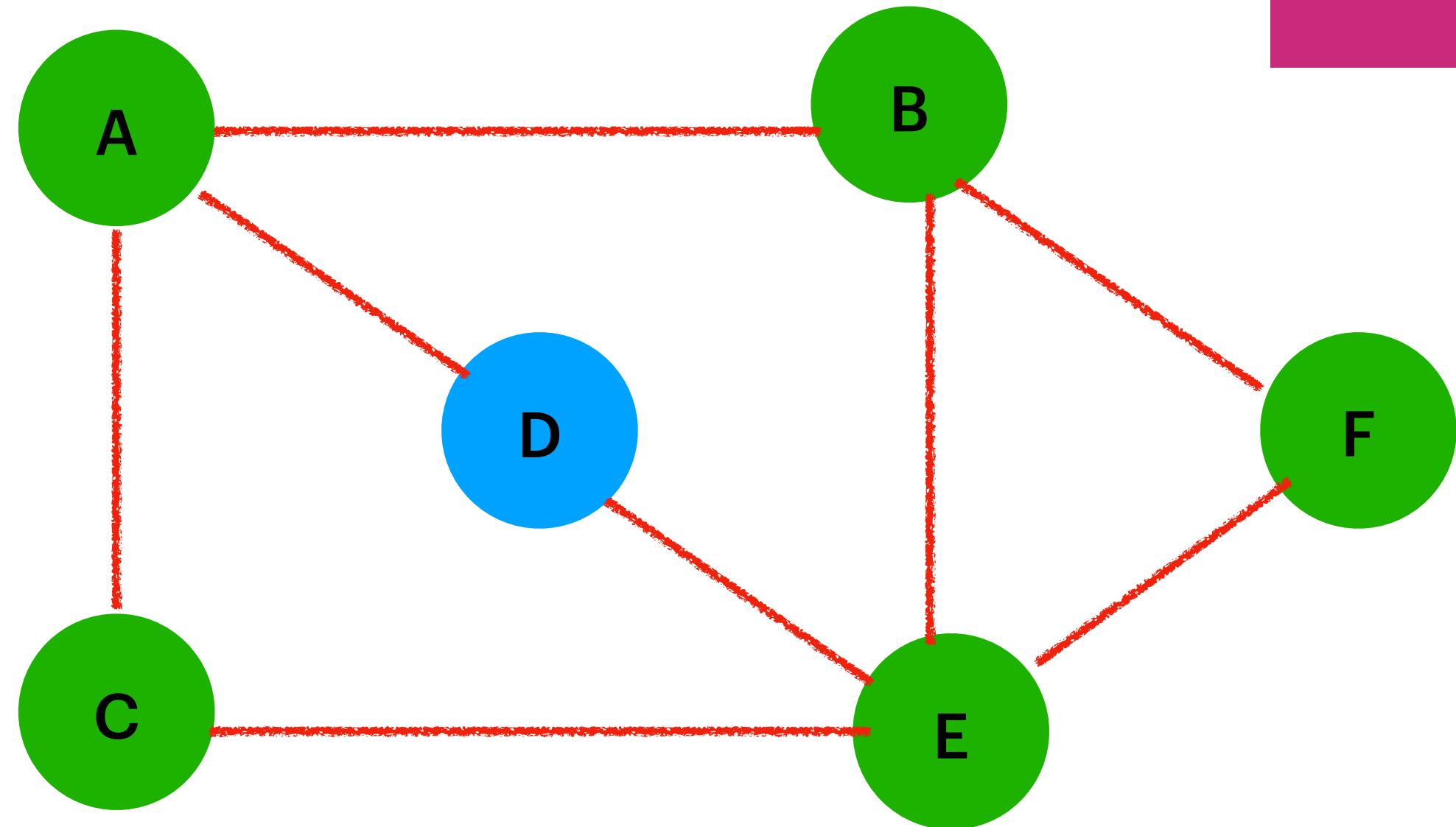


graph = [ [1,2], [2,3], [3], [] ] n=4



### Paths From 0 to 3

0  $\rightarrow$  1  $\rightarrow$  3 , 0  $\rightarrow$  2  $\rightarrow$  3 ,  
0  $\rightarrow$  1  $\rightarrow$  2  $\rightarrow$  3

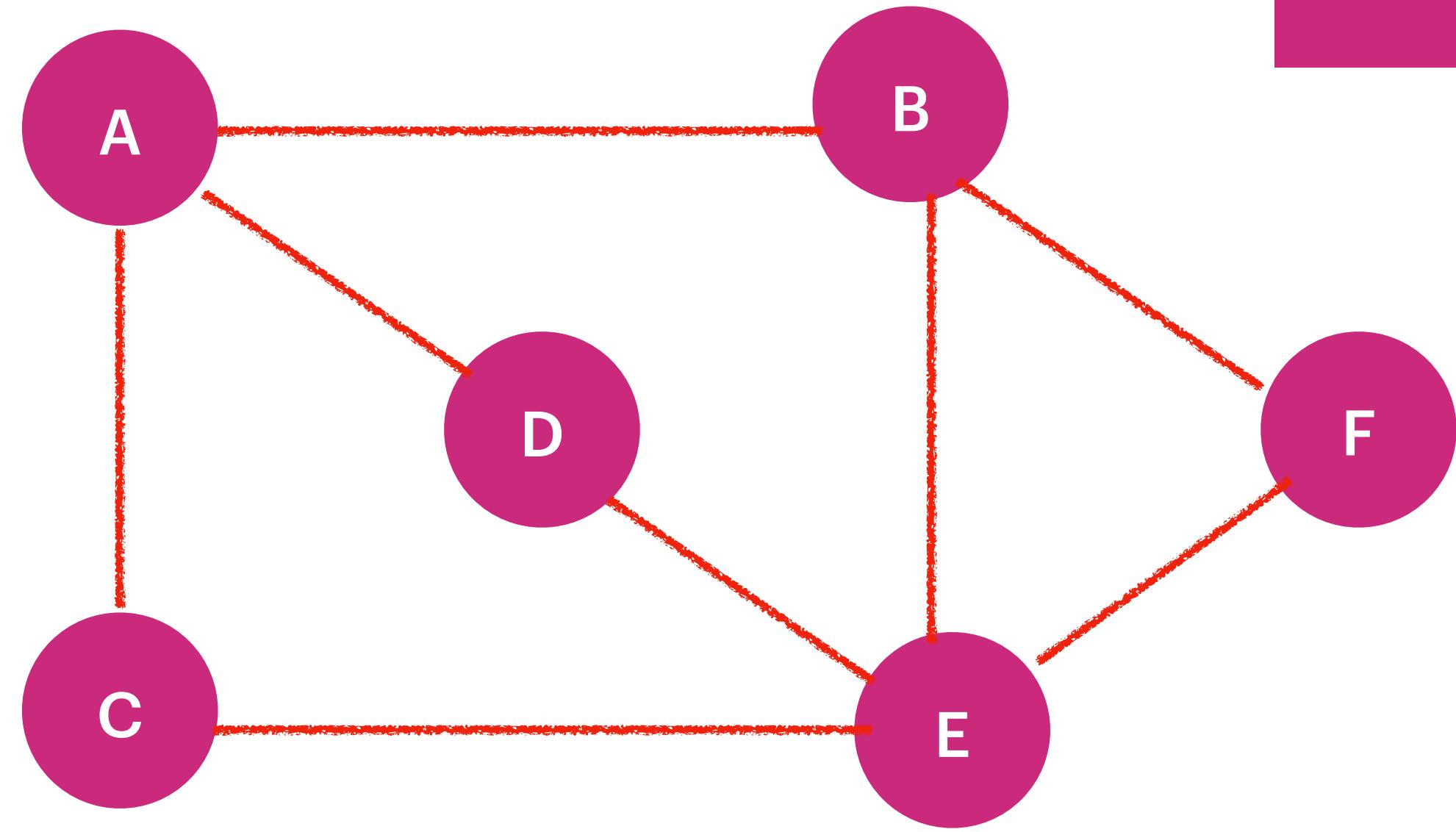


Print All the Paths From A  $\rightarrow$  F  
DFS :  $(V-1)!$   
BFS :  $(V-1)!$

BFS [  
[A, B, F],  
[A, C, E, F],  
[A, D, E, F],  
[A, B, E, F],  
[A, C, E, B, F],  
[A, D, E, B, F]  
]

output : [ [A,B,F], [A,B,E,F],[A,D,E,F],[A,D,E,B,F],[A,C,E,F],[A,C,E,B,F] ]

ABF, ACEBF, ADEF, ABEF, ACEBF, ADEBF



BFS A->F [A, B, F]  
 BFS B->D [B, A, D]  
 BFS E->A [E, B, A]  
 BFS C->B [C, A, B]  
 BFS E->F [E, F]

Shortest Path between A & F :  
 BFS :  $O(V+E)$   
 Output : [A,B,F]

A [C,D,B]  
 Queue [AC,AD,AB] V[A]  
 AC :  
 Queue [AD,AB,ACA,ACE] V[A,C]  
 AD:  
 Queue [AB,ACA,ACE,ADA,ADE] : V[A,C,D]  
 AB: V[A,C,D,B,E]  
 Queue [ABF,ACE[C,B,F]]  
 ABF -> F is Target Shortest Path

## Clone Graph

Given a reference of a node in a connected undirected graph.

Return a deep copy (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

```
class Node {  
    public int val;  
    public List<Node> neighbors;  
}
```

**Input:** adjList = [[2,4],[1,3],[2,4],[1,3]]

**Output:** [[2,4],[1,3],[2,4],[1,3]]

**Explanation:** There are 4 nodes in the graph.

1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

**Input:** adjList = [[2],[1]]

**Output:** [[2],[1]]

**Input:** adjList = []

**Output:** []

**Explanation:** Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.

Constraints:

The number of nodes in the graph is in the range [0, 100].

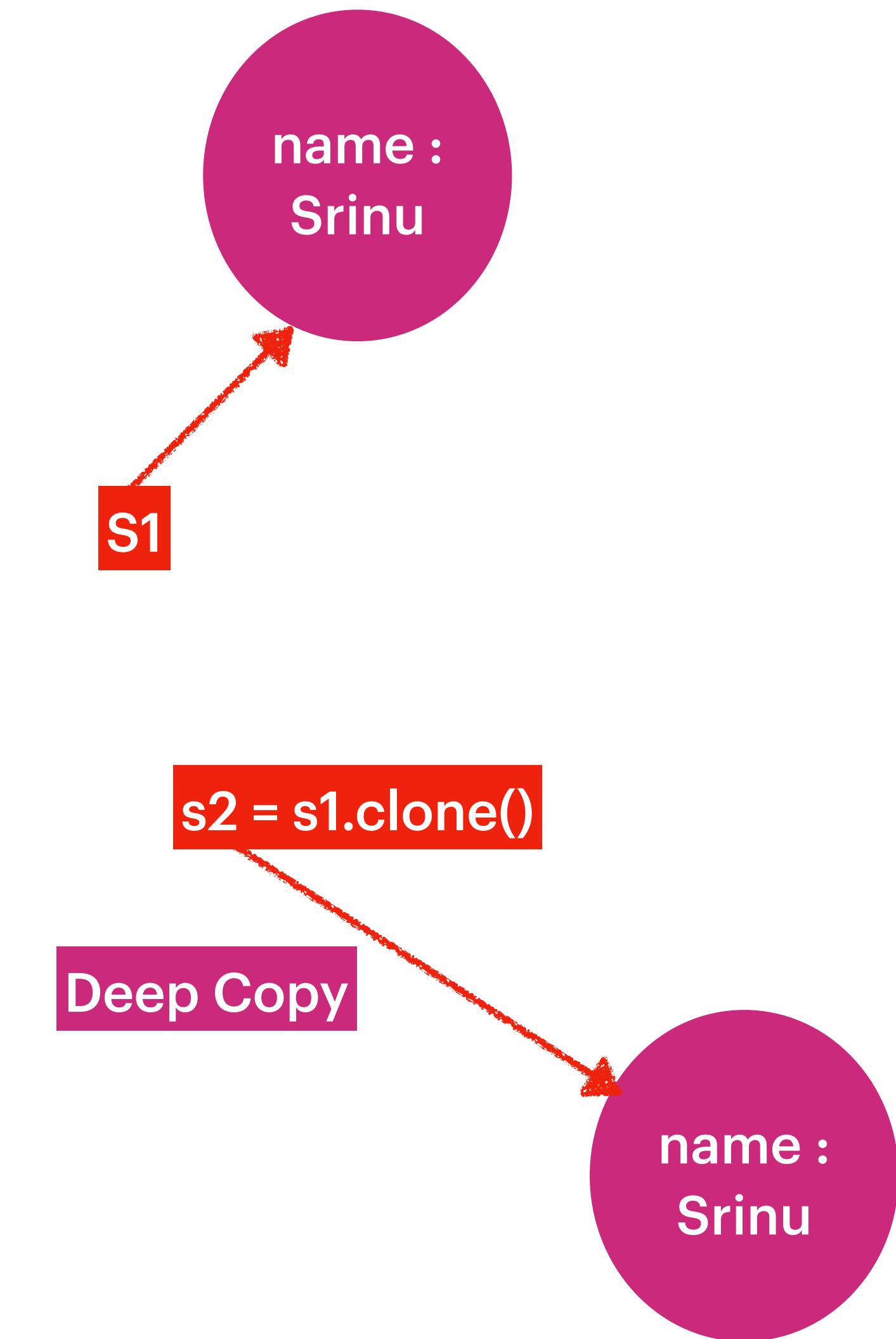
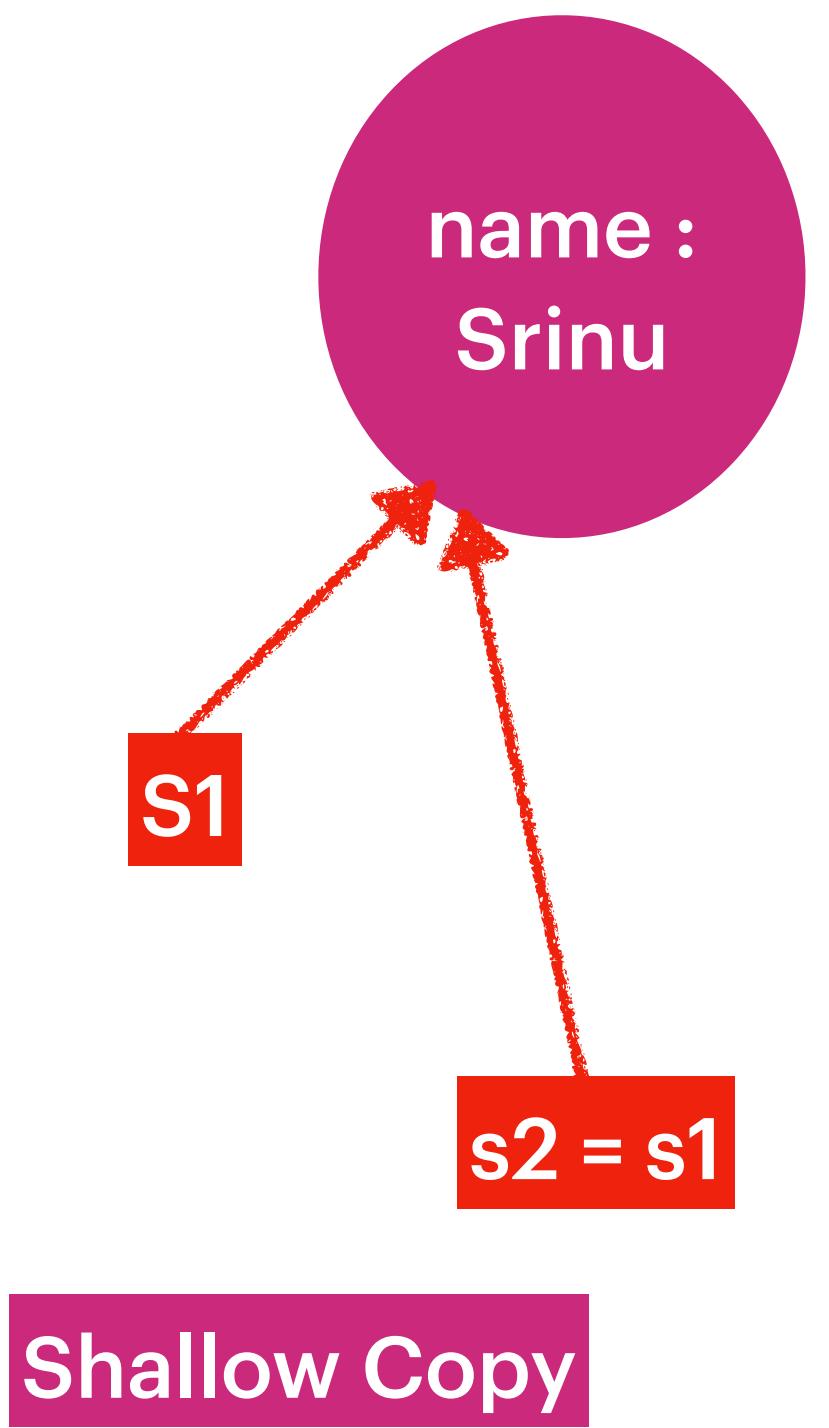
1 <= Node.val <= 100

Node.val is unique for each node.

There are no repeated edges  
and no self-loops in the graph.

The Graph is connected and all nodes can  
be visited starting from the given node.

```
class cloneGraph {  
    public Node cloneGraph(Node node) {  
        }  
    }
```



**Input:** adjList = [[2,4],[1,3],[2,4],[1,3]]

**Output:** [[2,4],[1,3],[2,4],[1,3]]

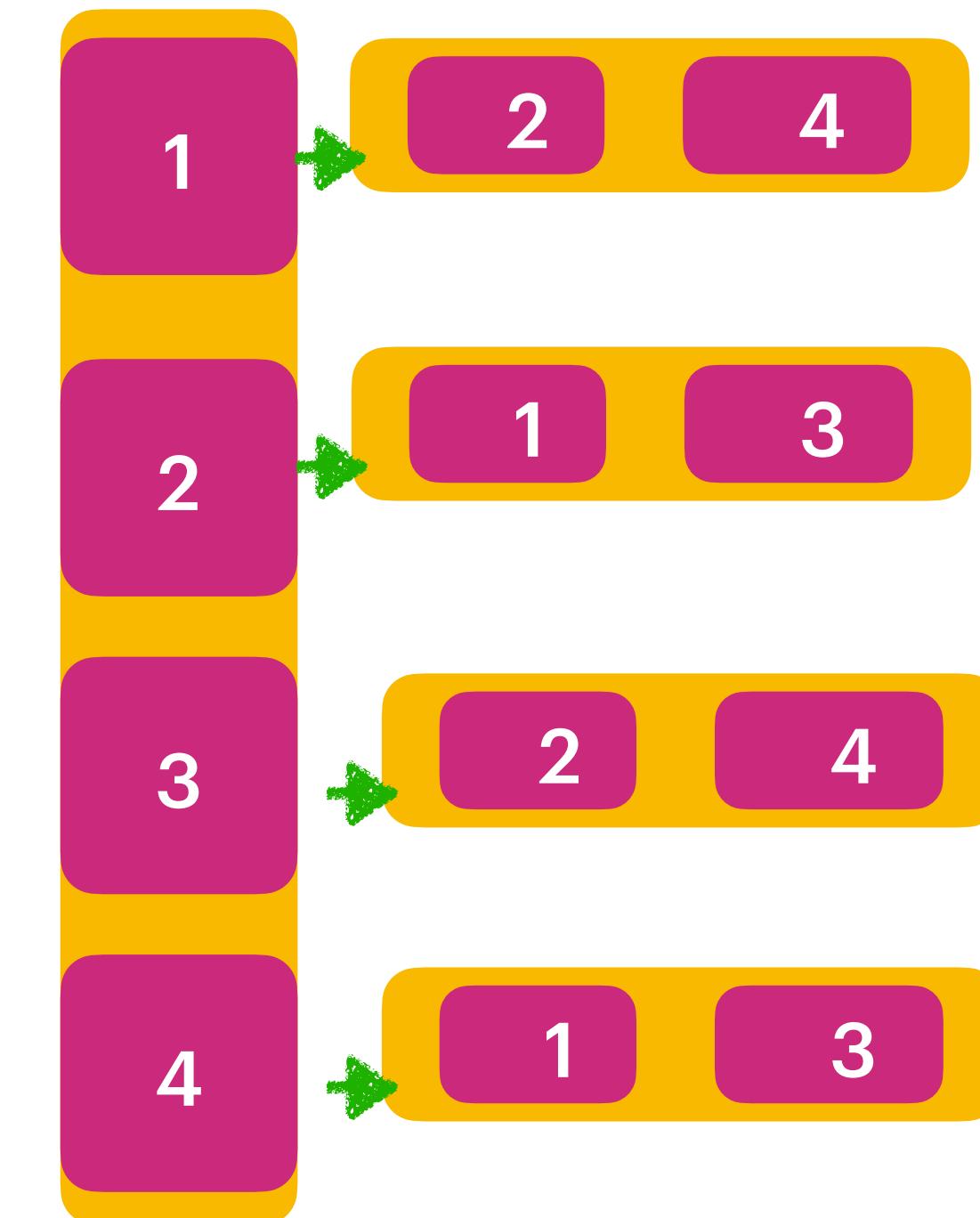
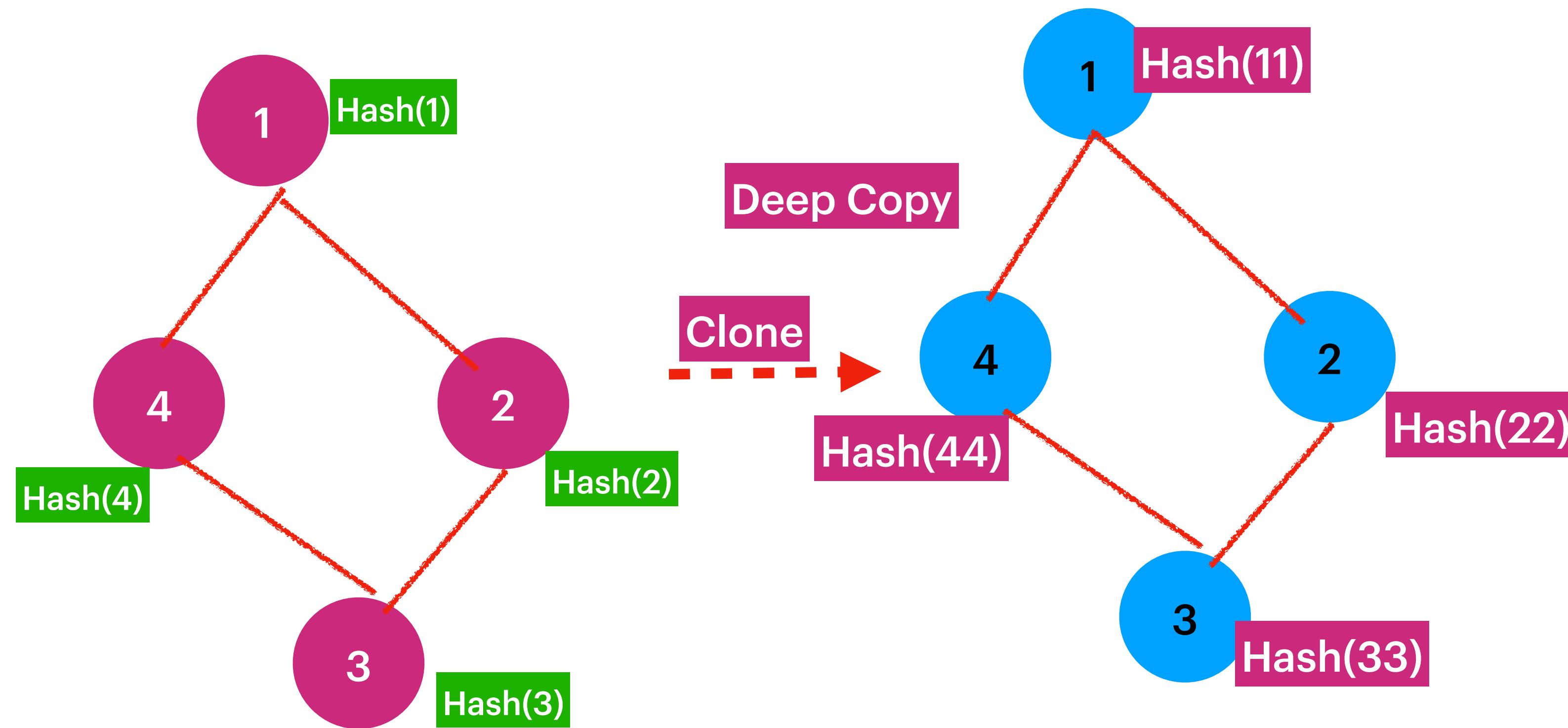
**Explanation:** There are 4 nodes in the graph.

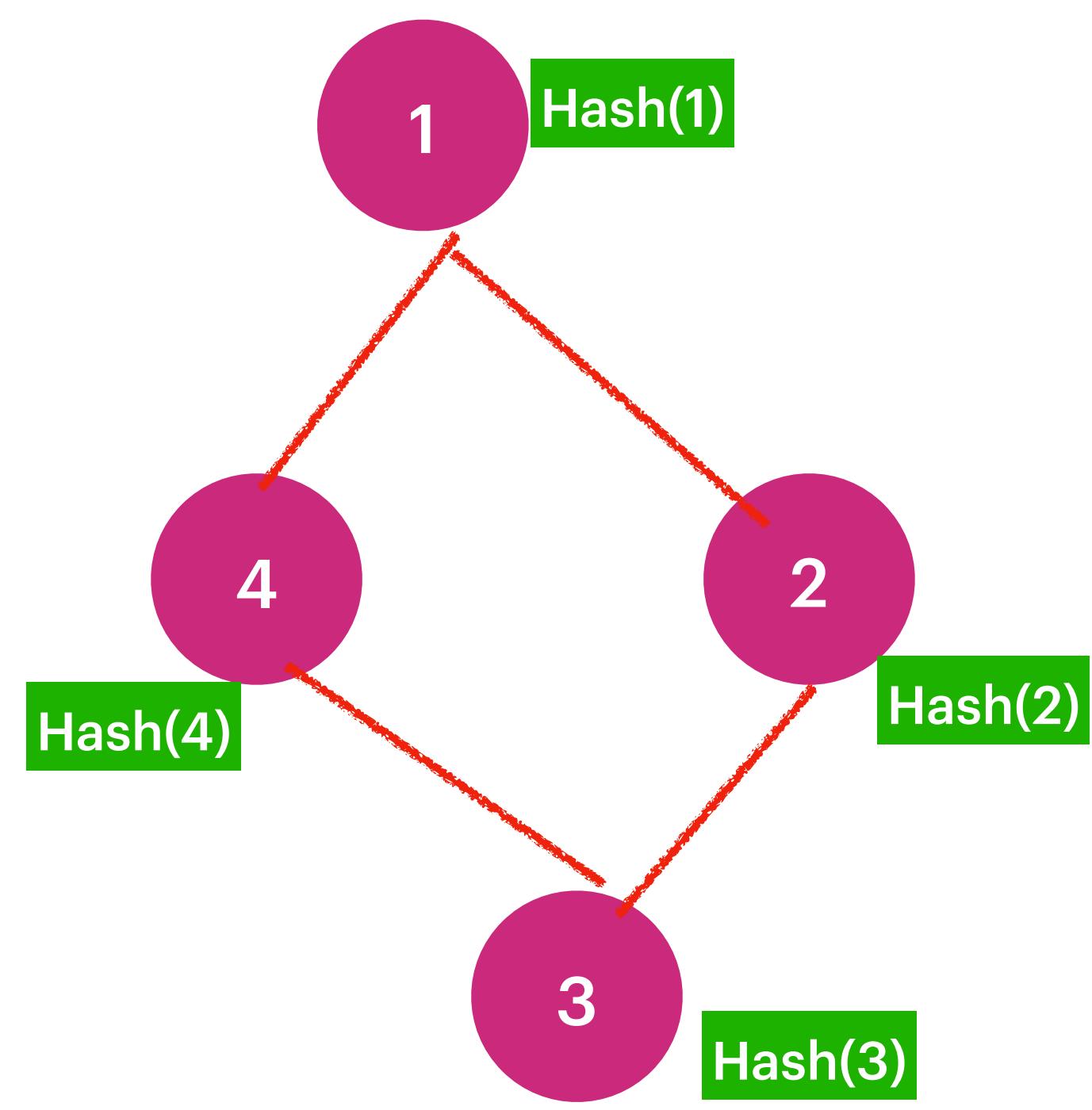
1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).





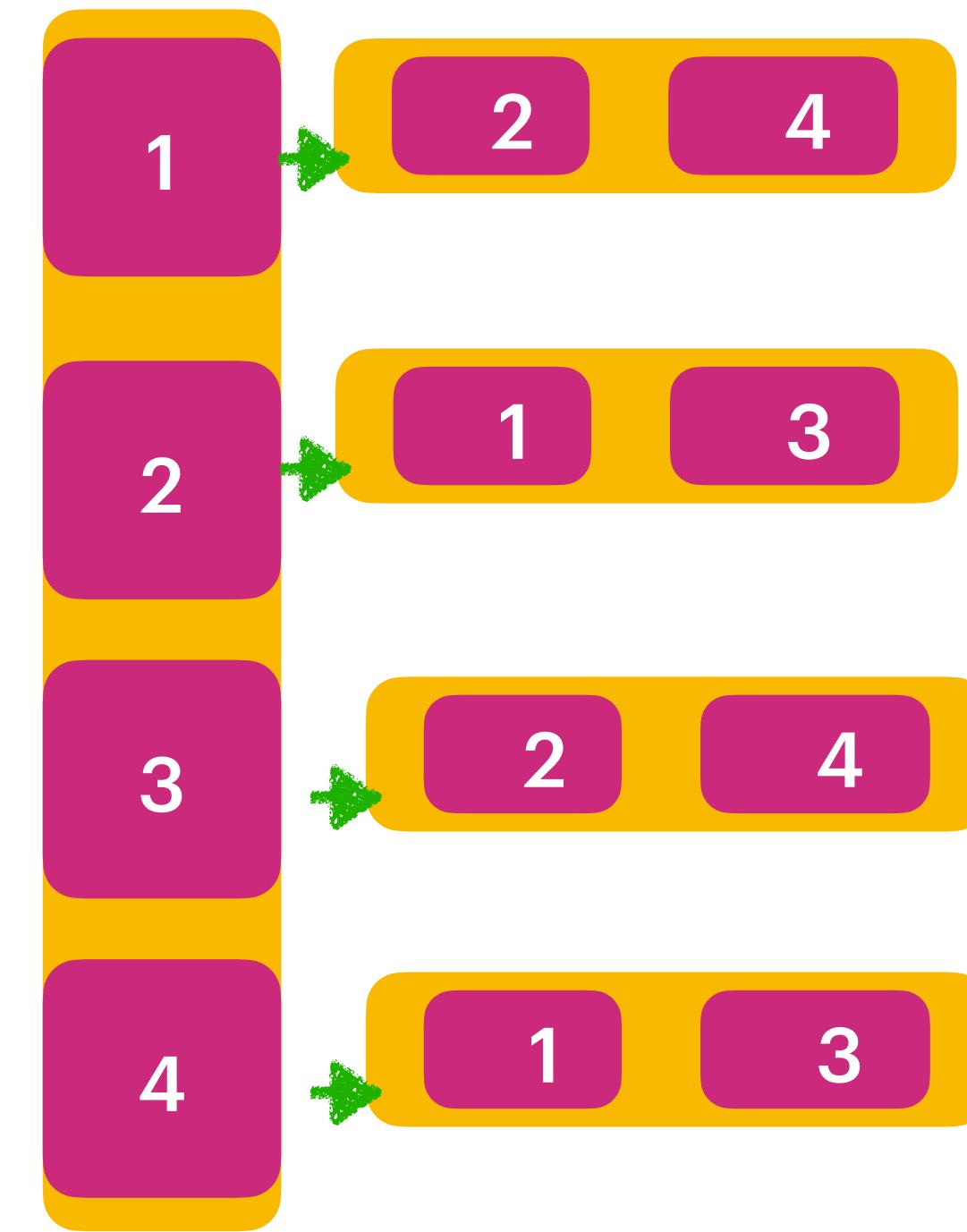
```

CloneNode(1)
  4: 2

CloneNode(4) :
1copyRefOfCloneNode(1) :
  3

CloneNode(2) :

```



## Construct N-ary Tree & Level Order Traversal

Given an n-ary tree, return the *level order* traversal of its nodes' values.

Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).

**Input:** root = [1,null,3,2,4,null,5,6]  
**Output:** [[1],[3,2,4],[5,6]]

**Input:** root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]  
**Output:** [[1],[2,3,4,5],[6,7,8,9,10],[11,12,13],[14]]

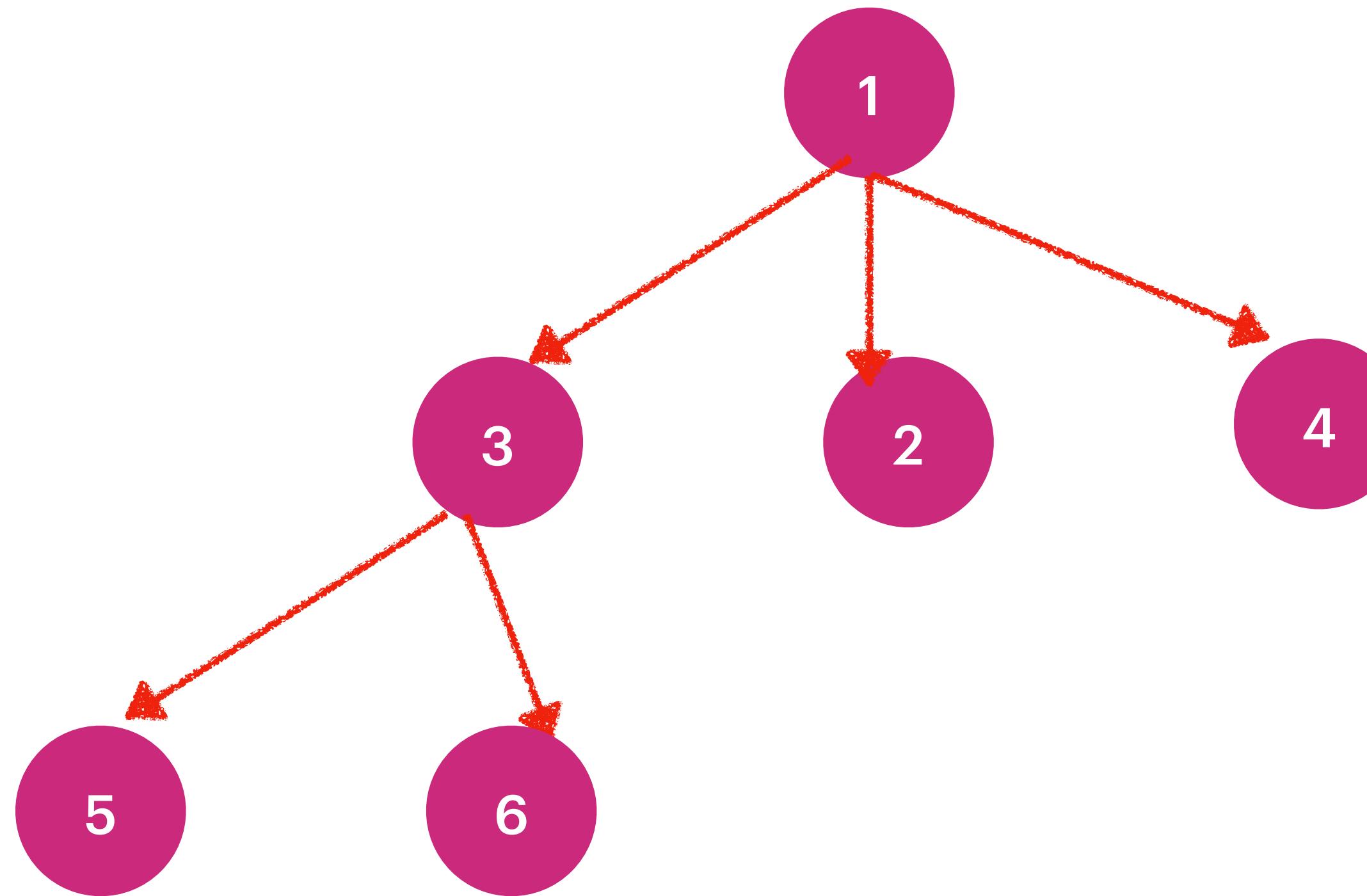
```
class Node {  
    public int val;  
    public List<Node> children;  
}
```

**Constraints:**  
The height of the n-ary tree is less than or equal to 1000  
The total number of nodes is between [0, 10<sup>4</sup>]

```
class Solution {  
public List<List<Integer>> levelOrder(Node root) {  
    }  
}
```

**Input:** root = [1,null,3,2,4,null,5,6]  
**Output:** [[1],[3,2,4],[5,6]]

### Construct N-ary Tree & Level Order Traversal

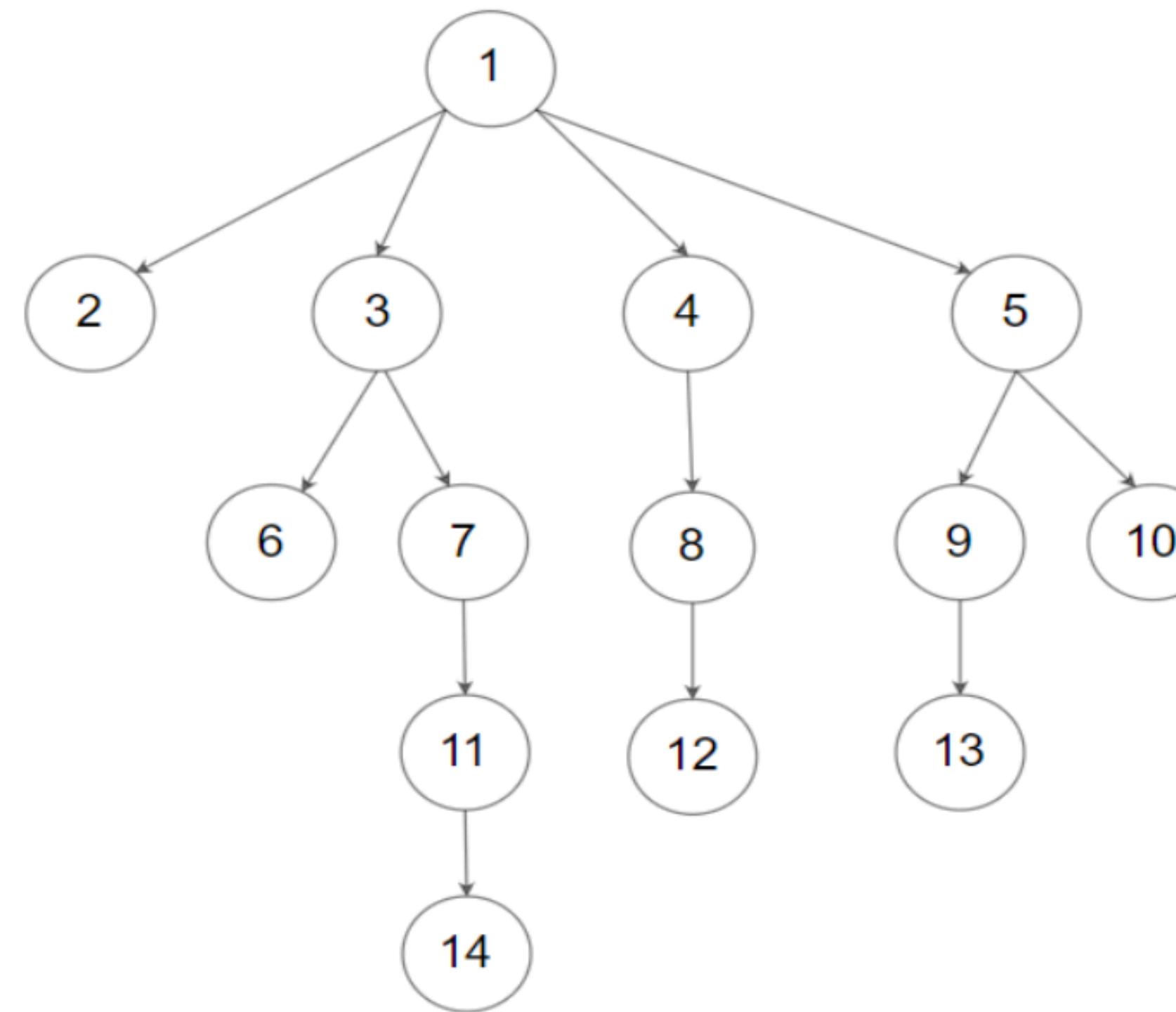


**Output:** [[1],[3,2,4],[5,6]]

**Input:** root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

**Output:** [[1],[2,3,4,5],[6,7,8,9,10],[11,12,13],[14]]

Queue[node[1])  
L1 [1]  
Queue : [2,3,4,5]  
  
L2[2,3,4,5]  
Queue[6,7,8,9,10]  
  
L3[6,7,8,9,10]  
Queue[11,12,13]  
  
L4 :[11,12,13]  
Queue: [14]  
  
L5[14]  
Queue:[]



**Construct N-ary Tree & Level Order Traversal**

**Output:** [[1],[2,3,4,5],[6,7,8,9,10],[11,12,13],[14]]

## Walls and Gates

You are given an  $m \times n$  grid rooms initialized with these three possible values.

-1 A wall or an obstacle.

0 A gate.

INF Infinity means an empty room.

We use the value  $2^{31} - 1 = 2147483647$  to represent INF as you may assume that the distance to a gate is less than 2147483647.

Fill each empty room with the distance to its nearest gate.

If it is impossible to reach a gate, it should be filled with INF.

Input: rooms = [

[2147483647,-1,0,2147483647],

[2147483647,2147483647,2147483647,-1],

[2147483647,-1,2147483647,-1],

[0,-1,2147483647,2147483647]

]

Output: [

[3,-1,0,1],

[2,2,1,-1],

[1,-1,2,-1],

[0,-1,3,4]

]

Input: rooms = [[-1]]

Output: [[-1]]

$m == \text{rooms.length}$

$n == \text{rooms}[i].length$

$1 \leq m, n \leq 250$

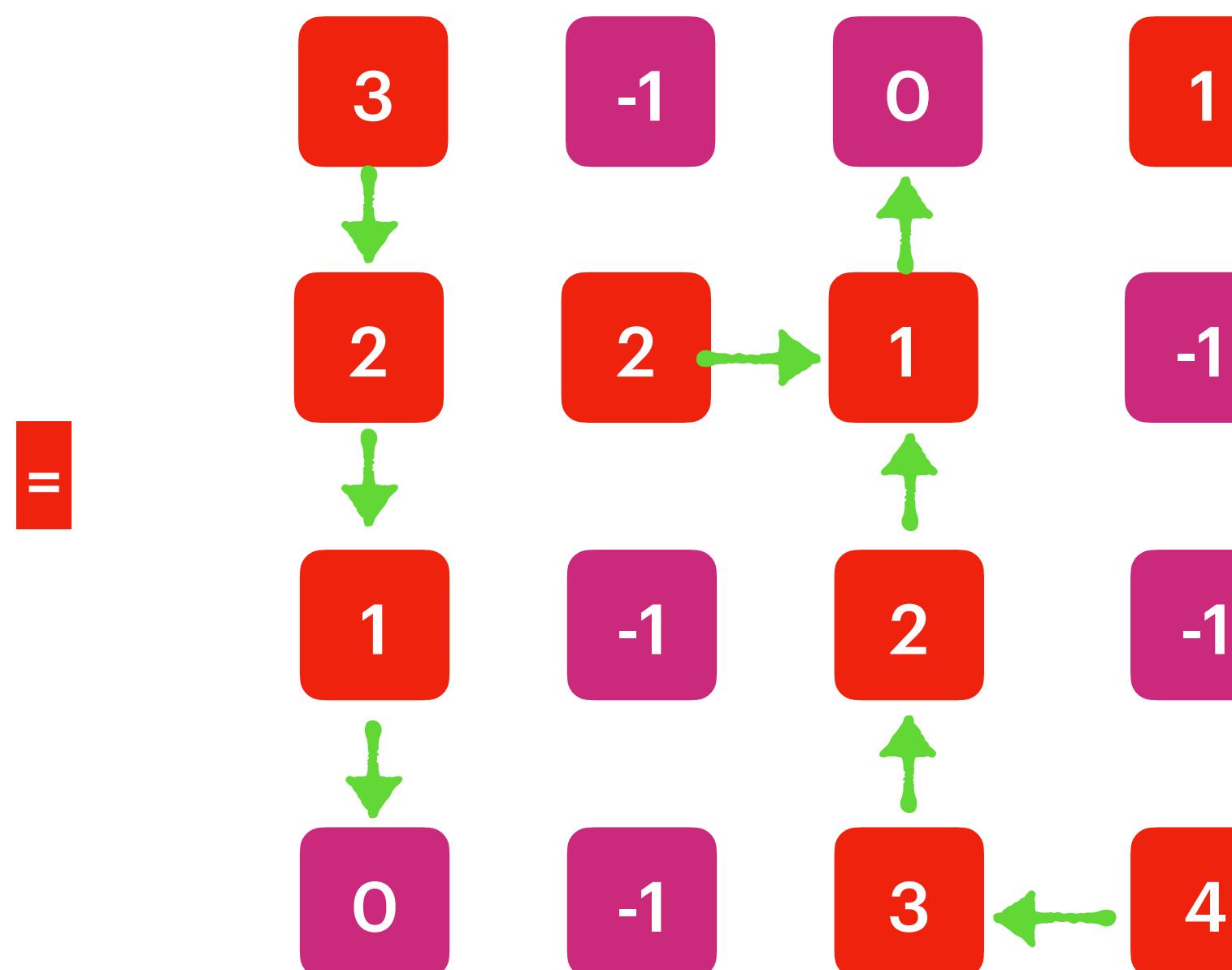
$\text{rooms}[i][j]$  is -1, 0, or  $2^{31} - 1$ .

```
[  
[E,-1,O,E],  
[E,E,E,-1],  
[E,-1,E,-1],  
[O,-1,E,E]  
]
```

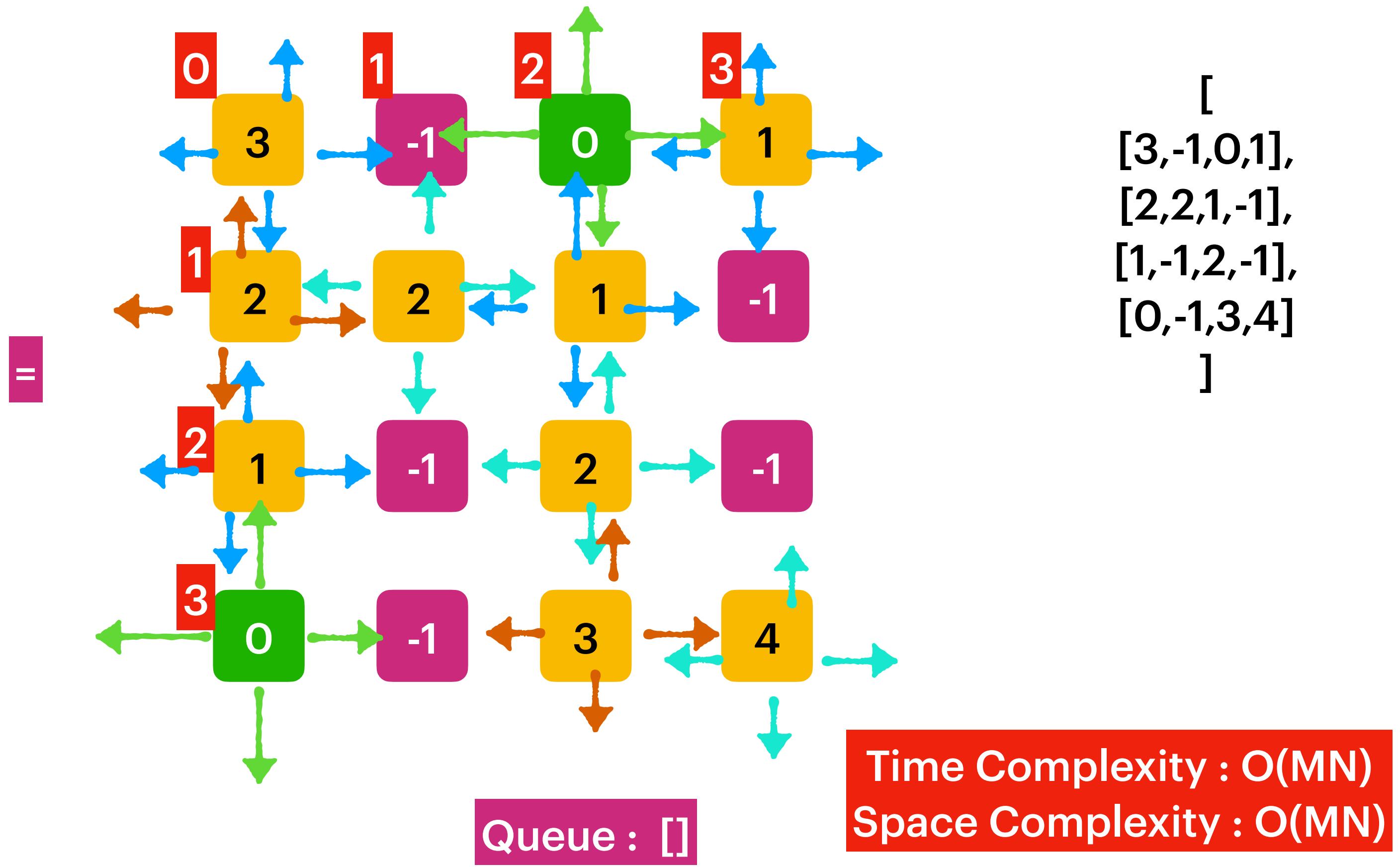
E = Integer.MAX\_VALUE = Empty Room  
-1 = Wall  
0 = Gate

Fill the Empty Room with nearest possible distance

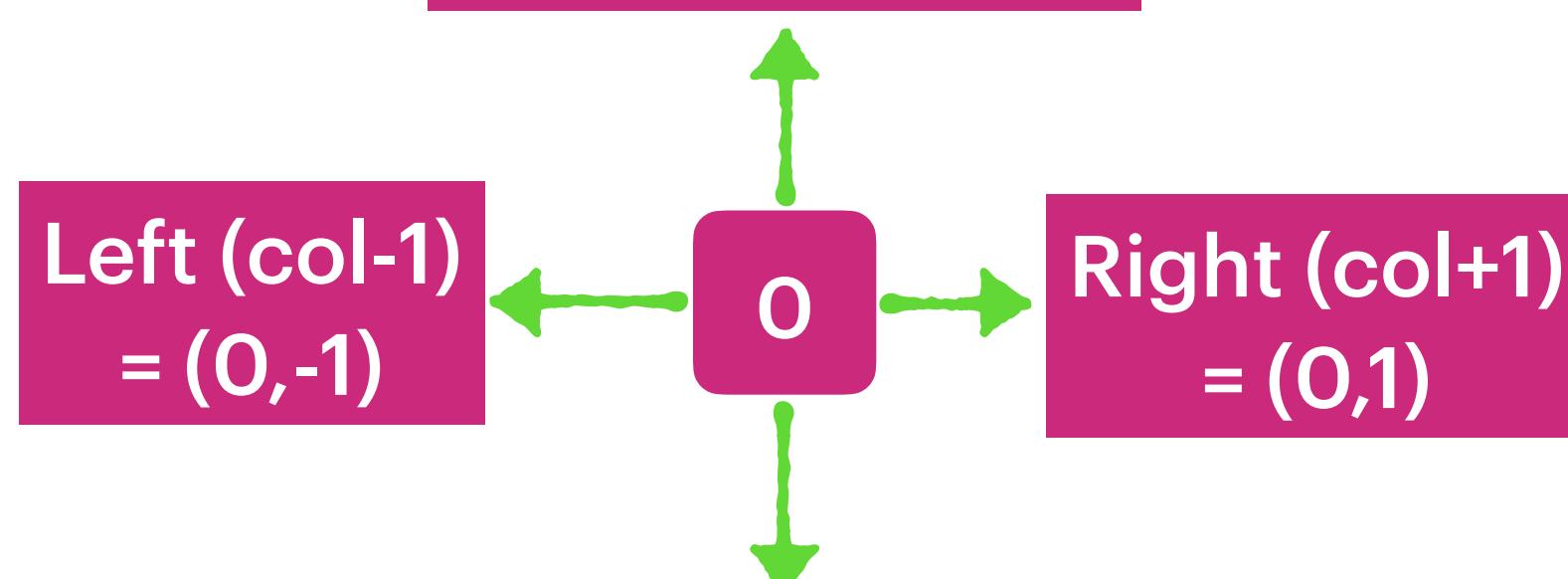
E	-1	0	E
E	E	E	-1
E	-1	E	-1
0	-1	E	E



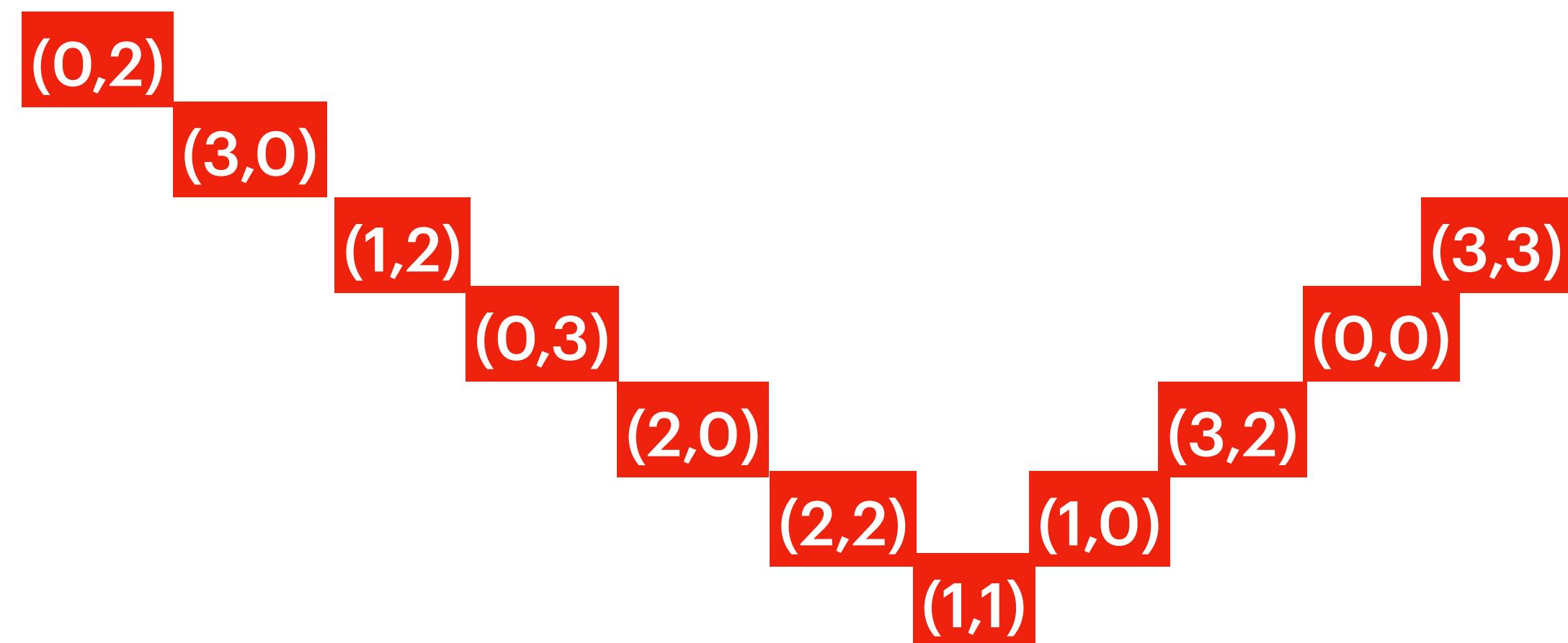
E	-1	0	E
E	E	E	-1
E	-1	E	-1
0	-1	E	E

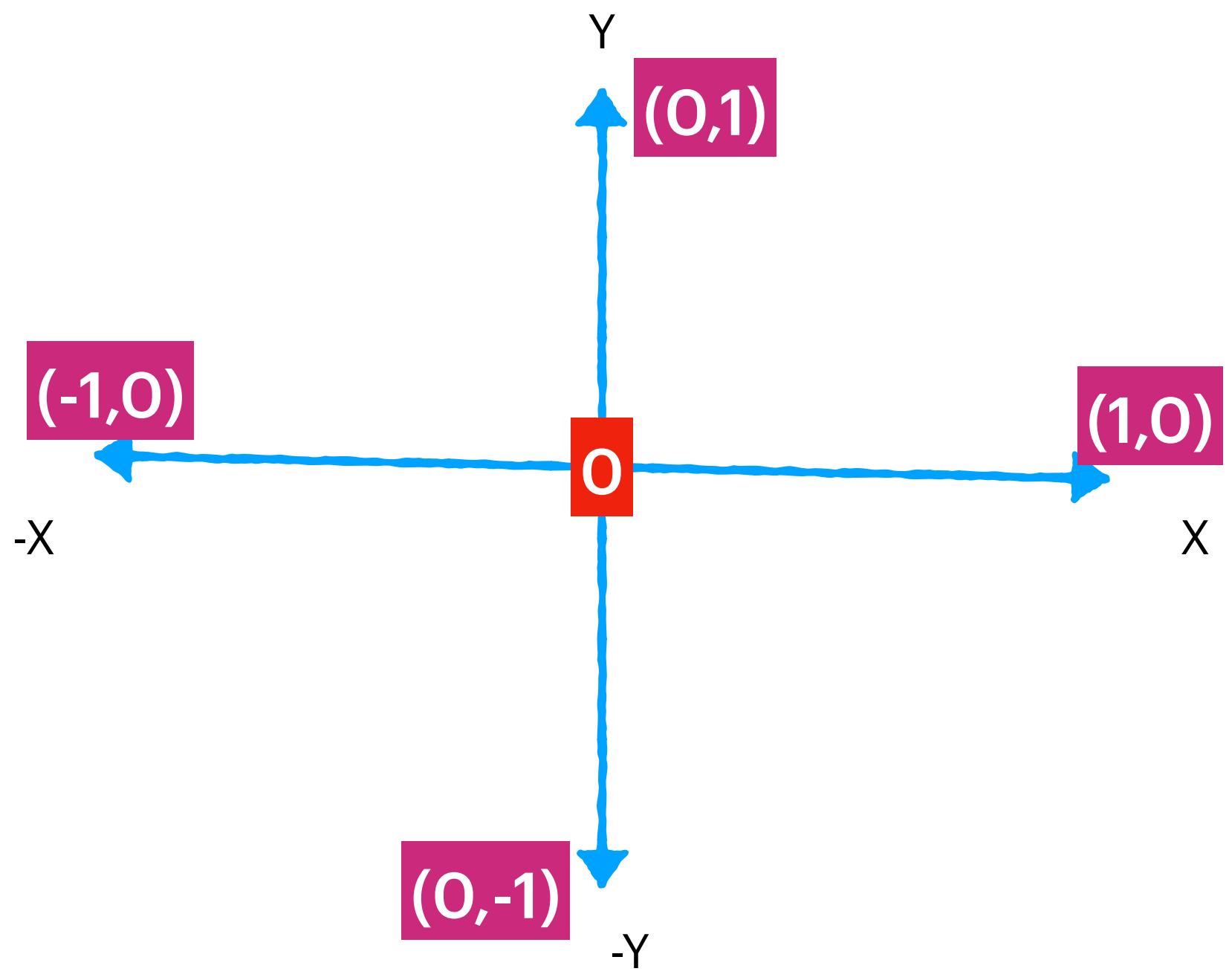


Up (row-1) = (-1,0)



Down (row+1) = (1,0)





## 01 Matrix

Given an  $m \times n$  binary matrix  $\text{mat}$ , return the distance of the nearest 0 for each cell.  
The distance between two adjacent cells is 1.

**Input:** mat = [  
  [0,0,0],  
  [0,1,0],  
  [0,0,0]  
]  
**Output:** [  
  [0,0,0],  
  [0,1,0],  
  [0,0,0]  
]

**Input:** mat = [  
  [0,0,0],  
  [0,1,0],  
  [1,1,1]  
]  
**Output:** [  
  [0,0,0],  
  [0,1,0],  
  [1,2,1]  
]

**Constraints:**  
 $m == \text{mat.length}$   
 $n == \text{mat[i].length}$   
 $1 \leq m, n \leq 104$   
 $1 \leq m * n \leq 104$   
 $\text{mat}[i][j]$  is either 0 or 1.  
There is at least one 0 in mat.

0	0	0
0	1	0
0	0	0

0	0	0
0	1	0
0	0	0

=

Input: mat = [  
[0,0,0],  
[0,1,0],  
[0,0,0]  
]  
Output: [  
[0,0,0],  
[0,1,0],  
[0,0,0]  
]

0	0	0
0	1	0
1	1	1

Input: mat = [  
[0,0,0],  
[0,1,0],  
[1,1,1]  
]  
Output: [  
[0,0,0],  
[0,1,0],  
[0,0,0]  
]

1 1 1

1 1 1

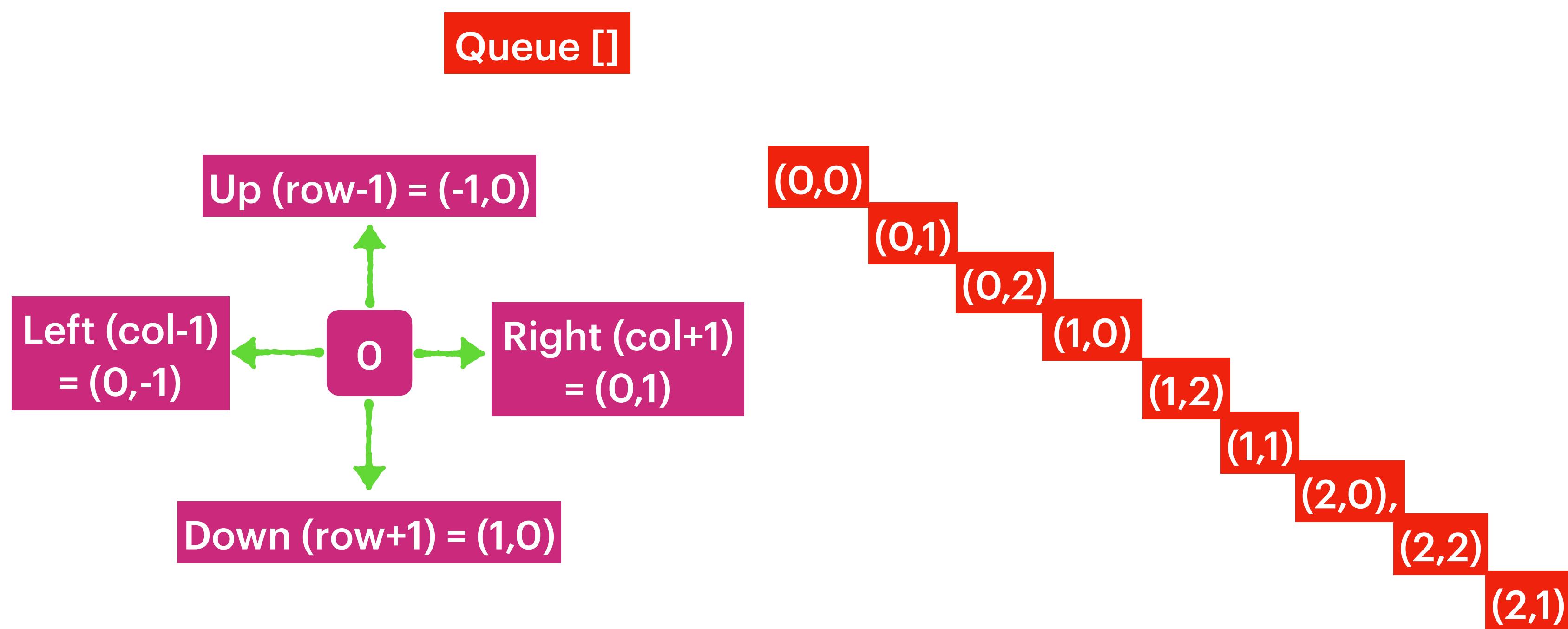
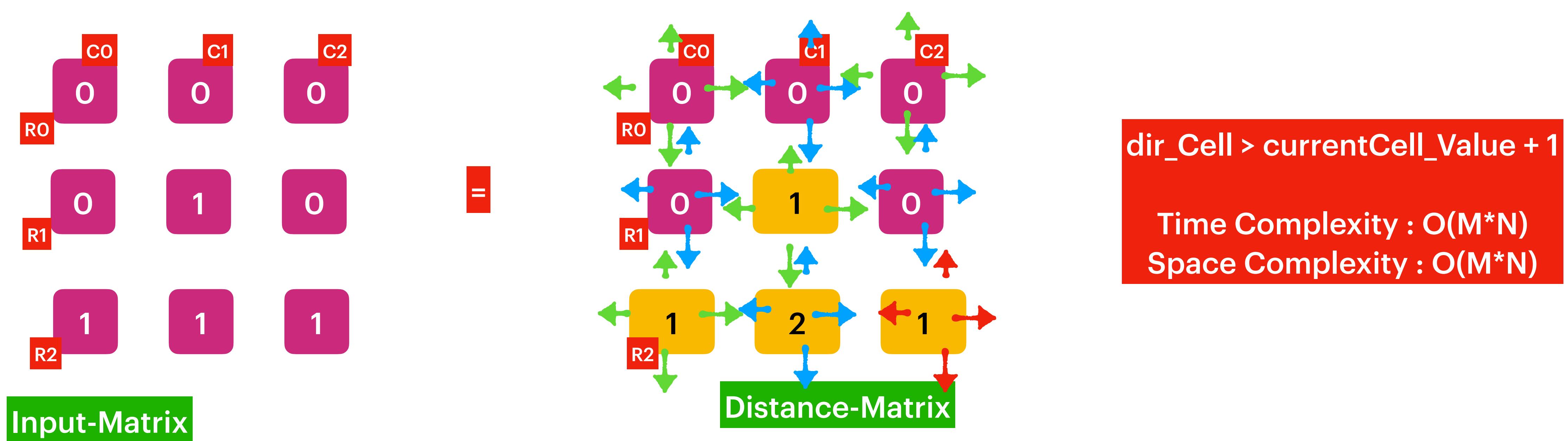
1 0 1

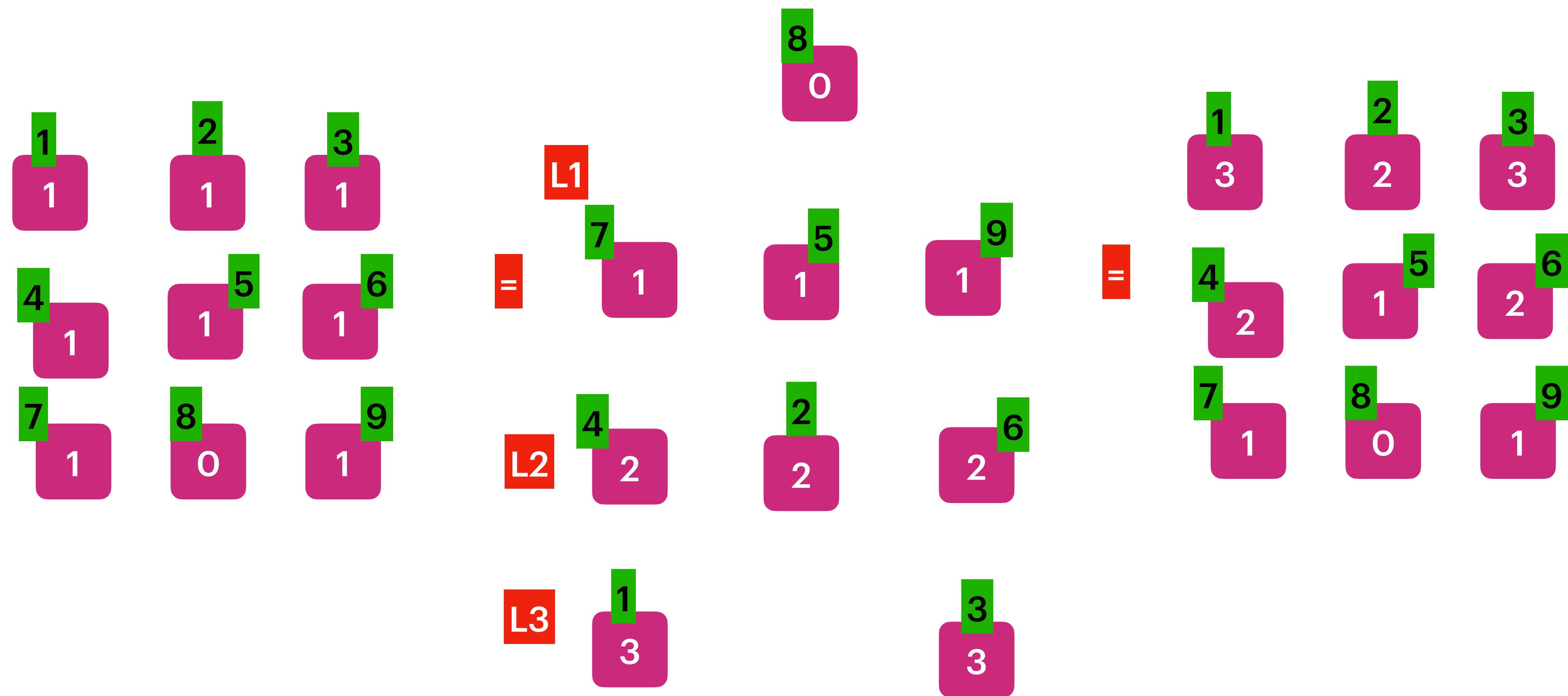
3 2 3

2 1 2

1 0 1

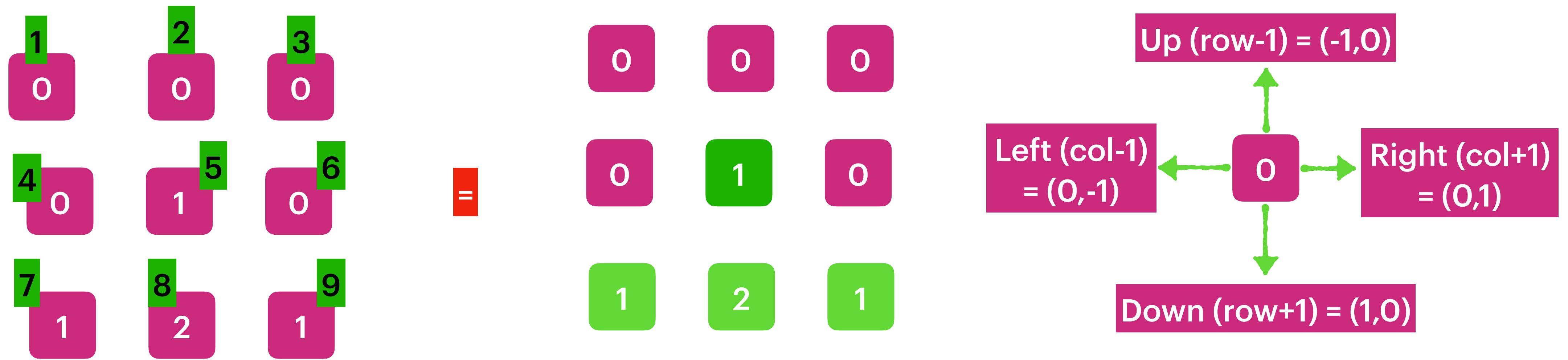
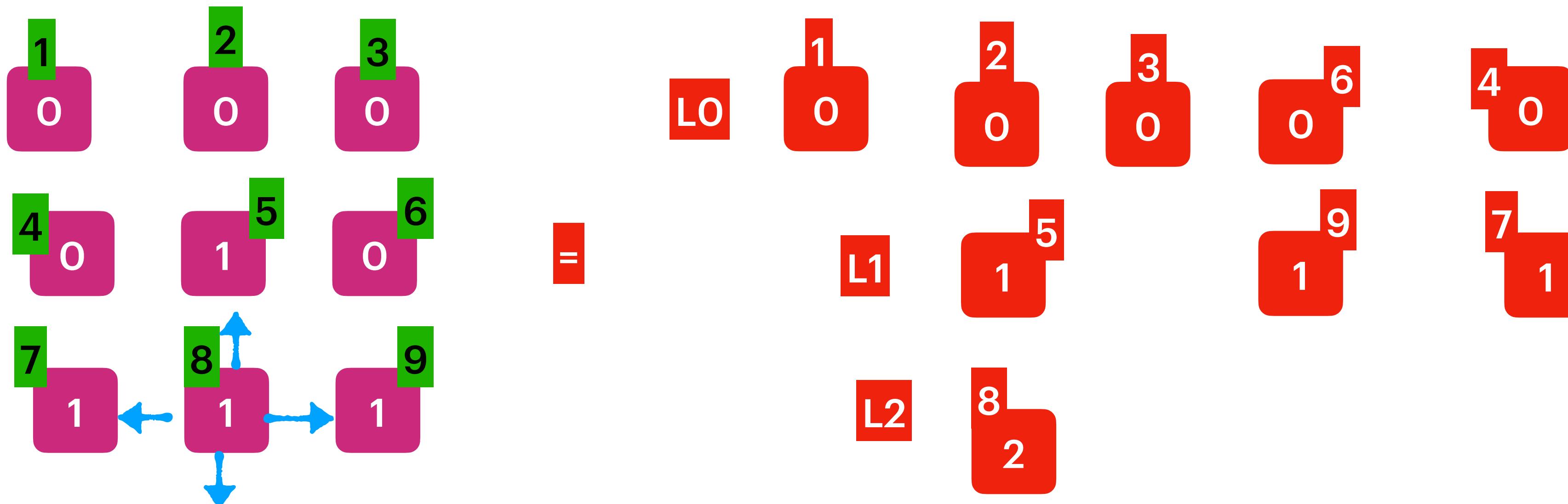
=





**Output :**

3	2	3
2	1	2
1	0	1



## Rotting Oranges

You are given an  $m \times n$  grid where each cell can have one of three values:

- 0 representing an empty cell,
- 1 representing a fresh orange, or
- 2 representing a rotten orange.

Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange.  
If this is impossible, return -1.

**Input:** grid = [[2,1,1],[1,1,0],[0,1,1]]  
**Output:** 4

**Input:** grid = [[2,1,1],[0,1,1],[1,0,1]]  
**Output:** -1

**Explanation:** The orange in the bottom left corner  
(row 2, column 0) is never rotten,  
because rotting only happens 4-directionally.

**Input:** grid = [[1]]  
**Output:** -1

**Explanation:** Since the orange can never rotate.

**Input:** grid = [[0,0,0,0,0]]  
**Output:** 0

**Explanation:** Since there are no oranges to rotate.

**Input:** grid = [[0]]  
**Output:** 0

**Explanation:** Since there are already  
no fresh oranges at minute 0,  
the answer is just 0.

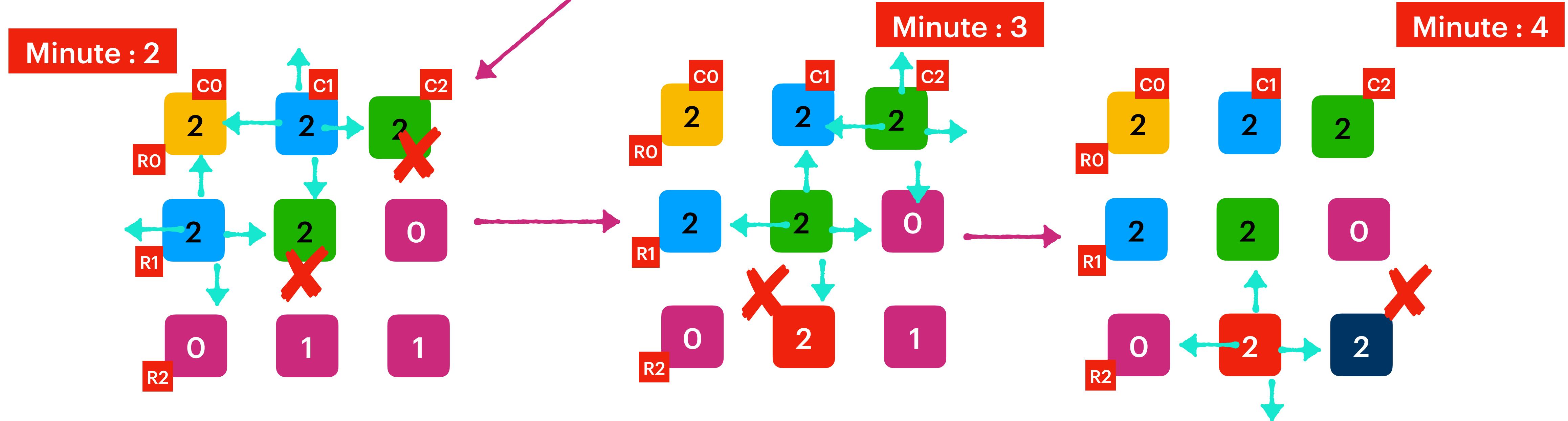
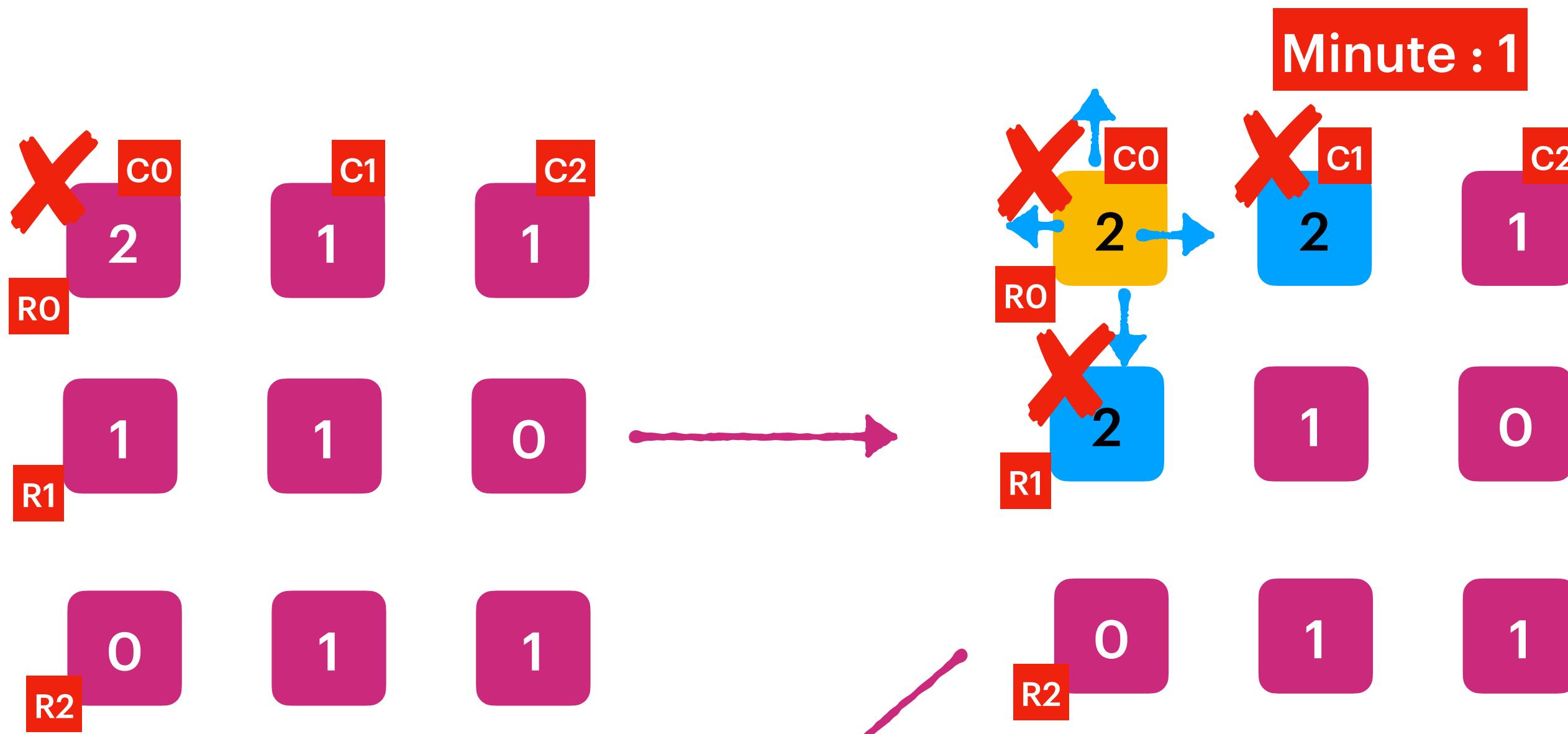
**Constraints:**

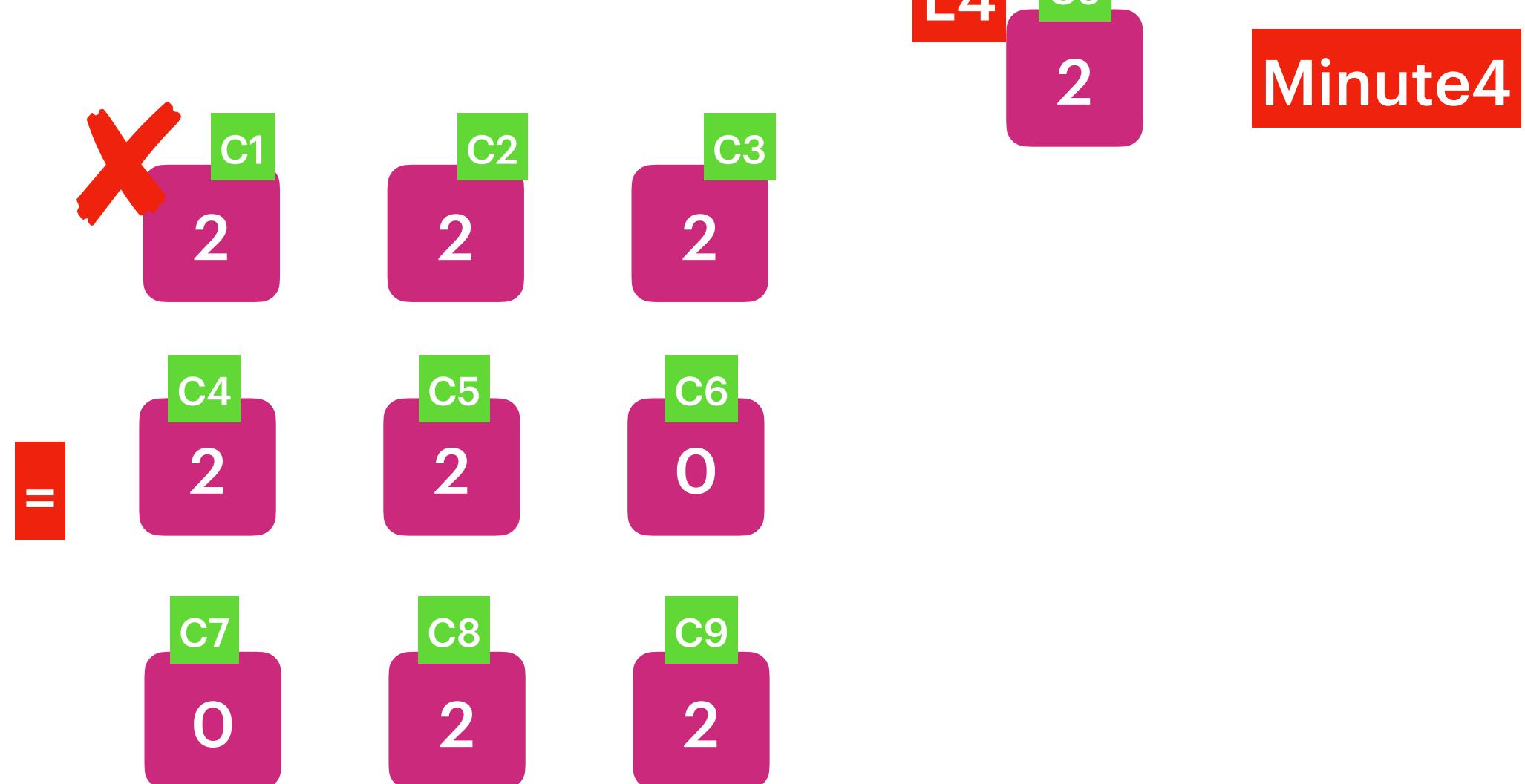
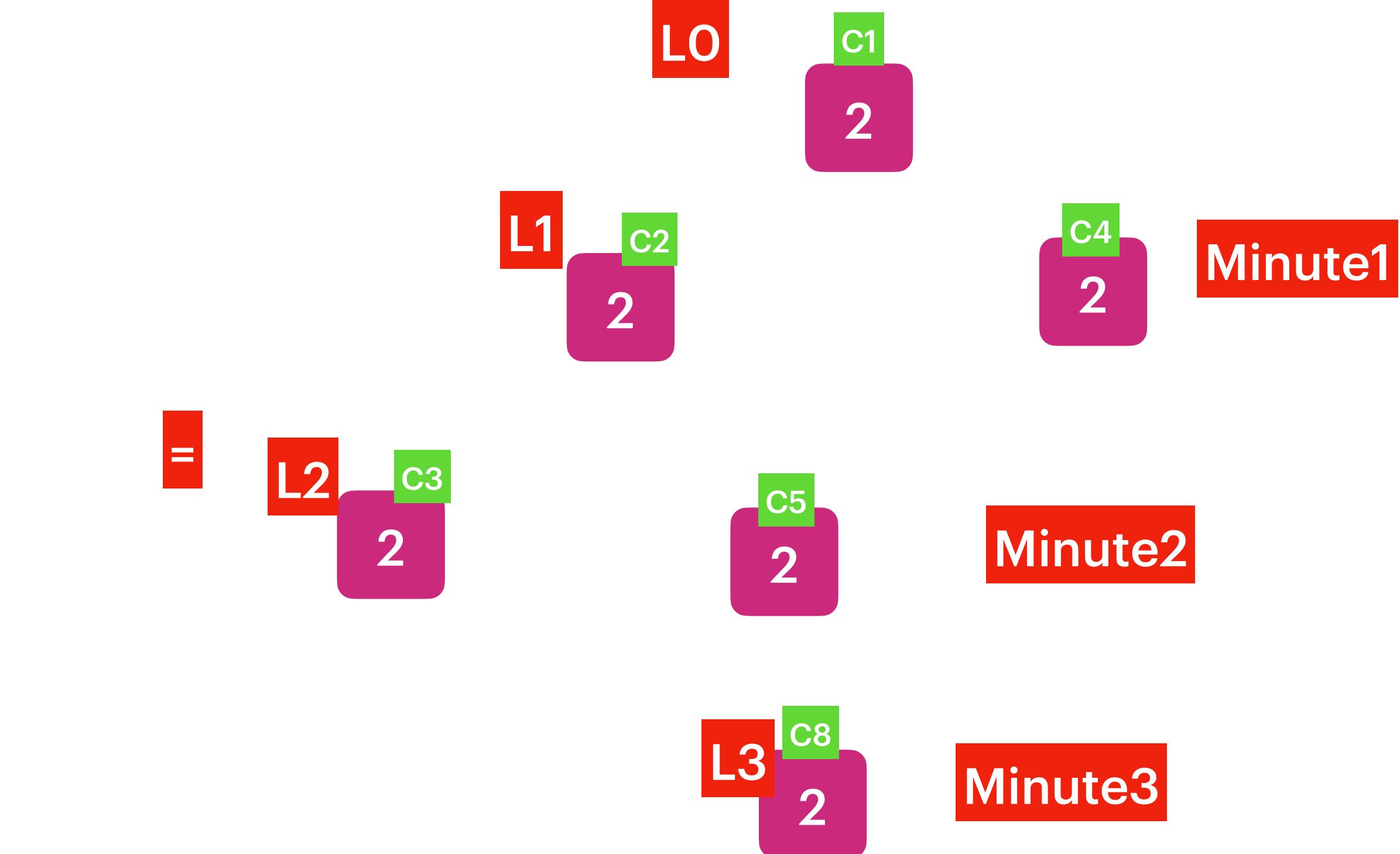
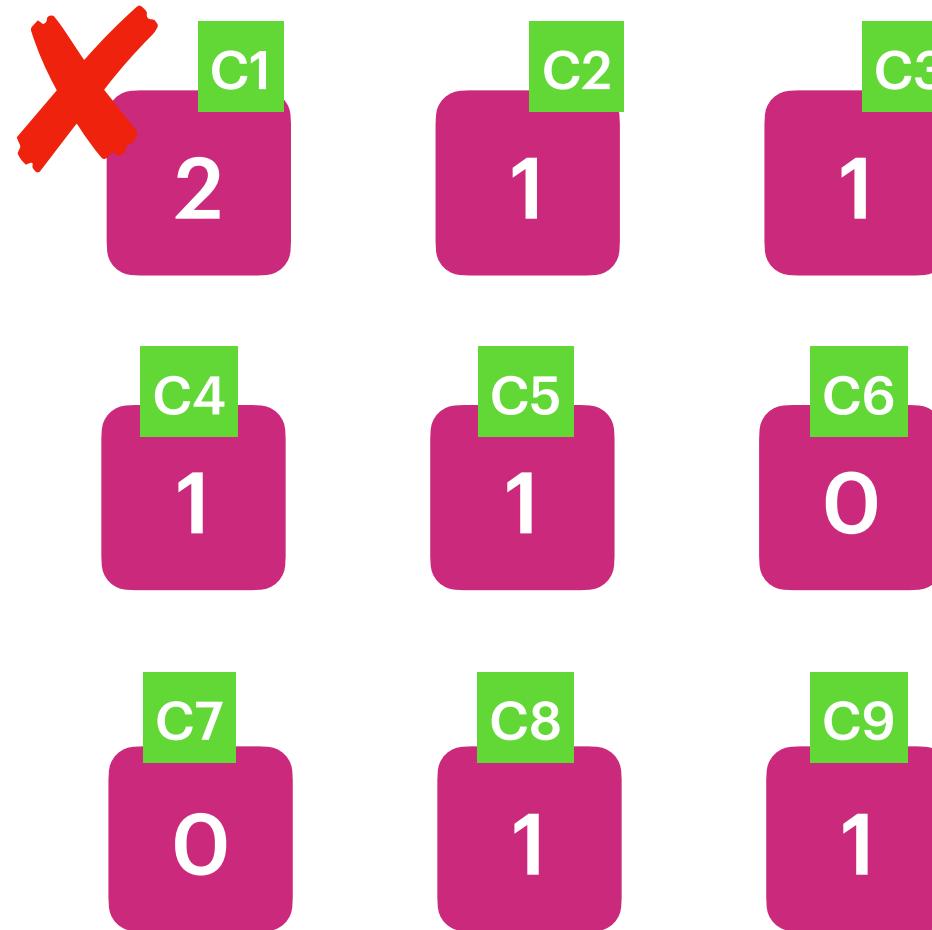
$m == \text{grid.length}$   
 $n == \text{grid[i].length}$   
 $1 \leq m, n \leq 10$   
 $\text{grid[i][j]}$  is 0, 1, or 2.

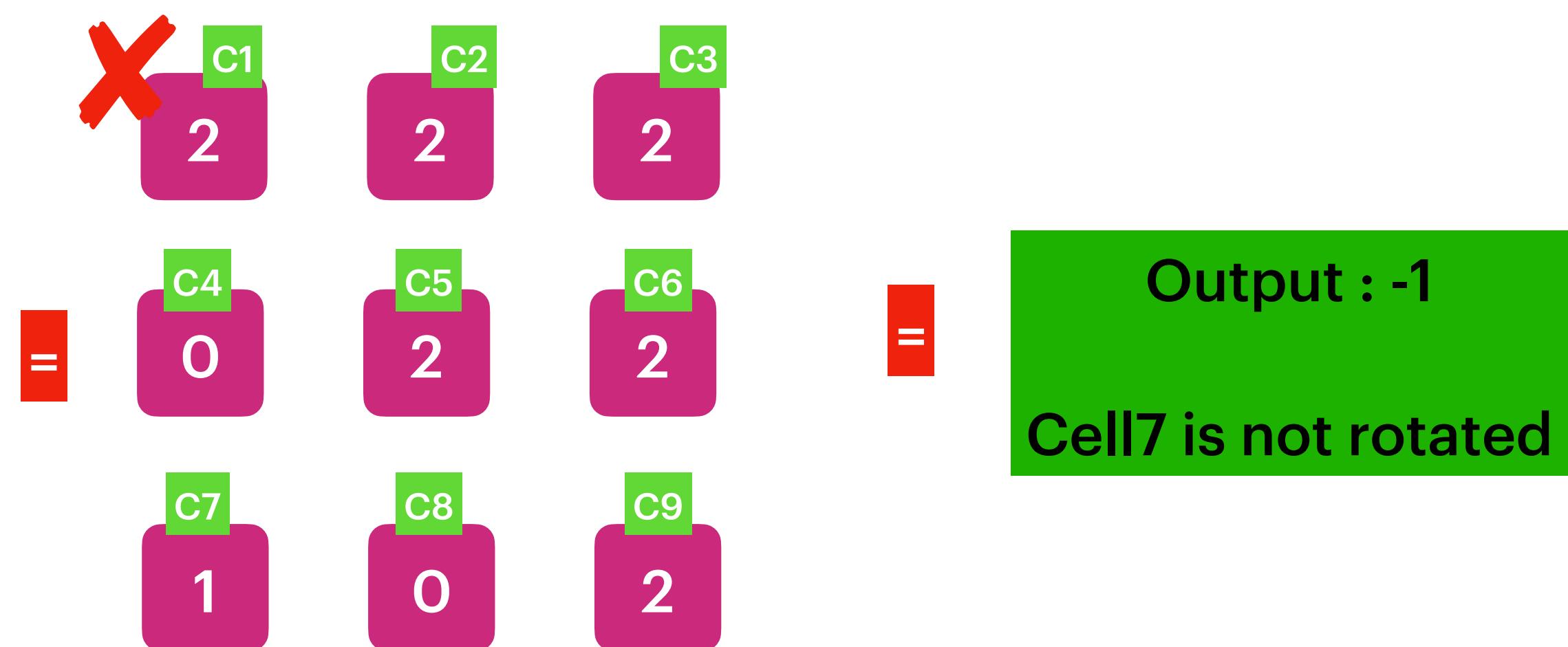
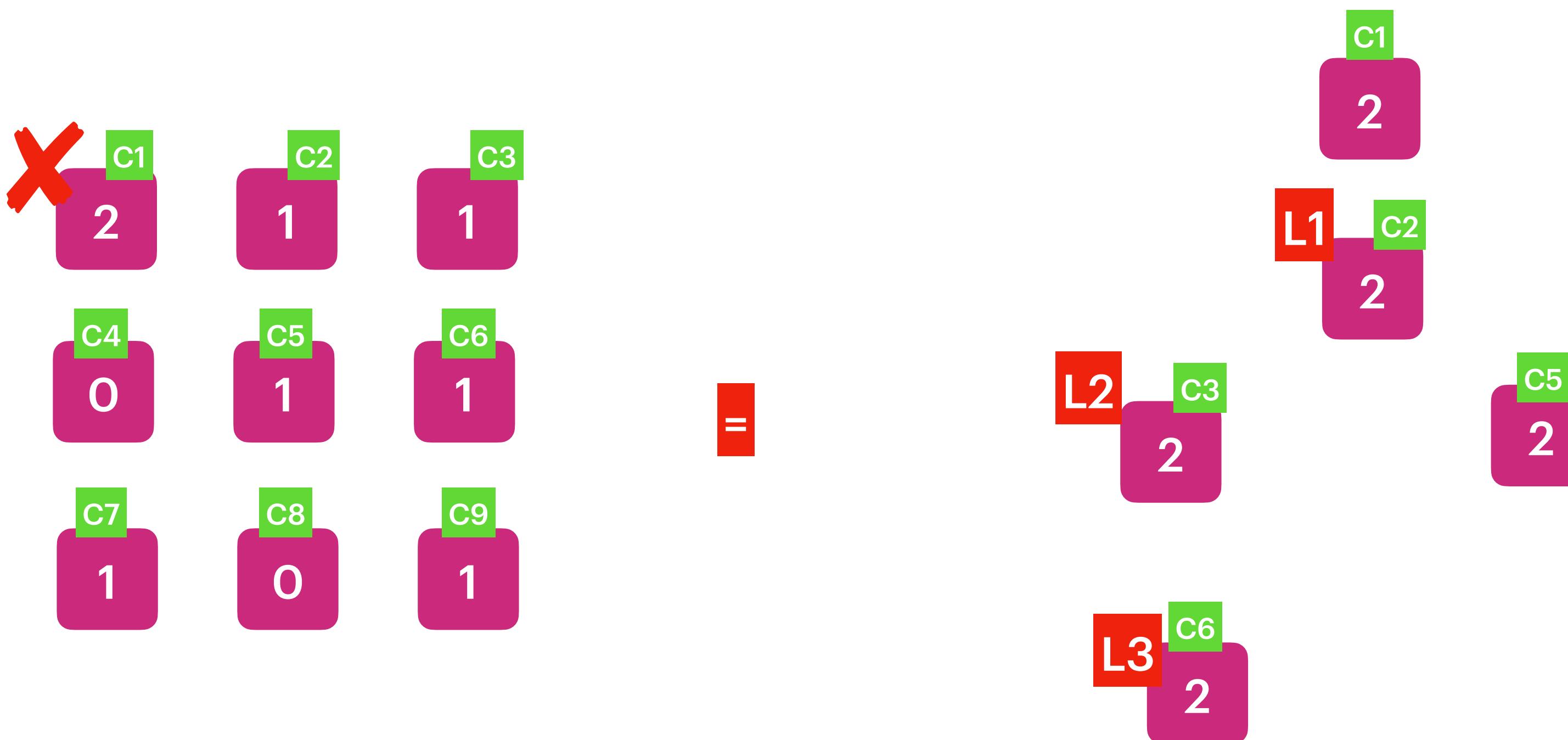
**Input:** grid = [[0,2]]  
**Output:** 0

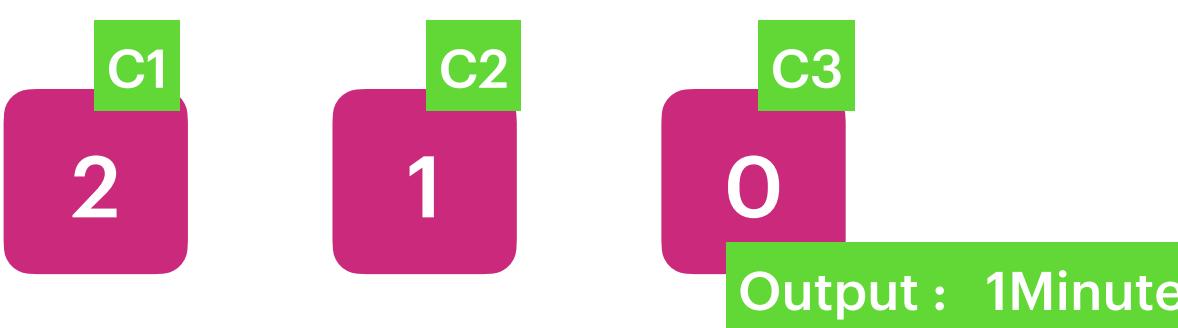
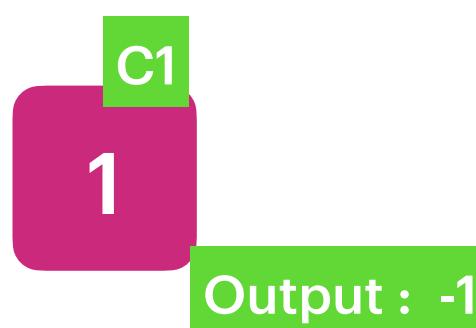
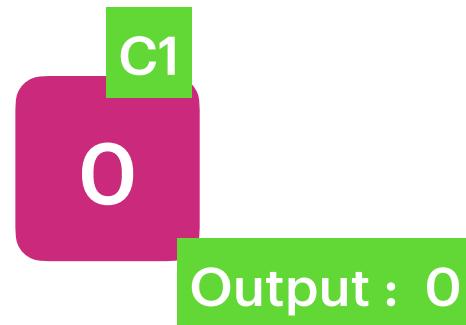
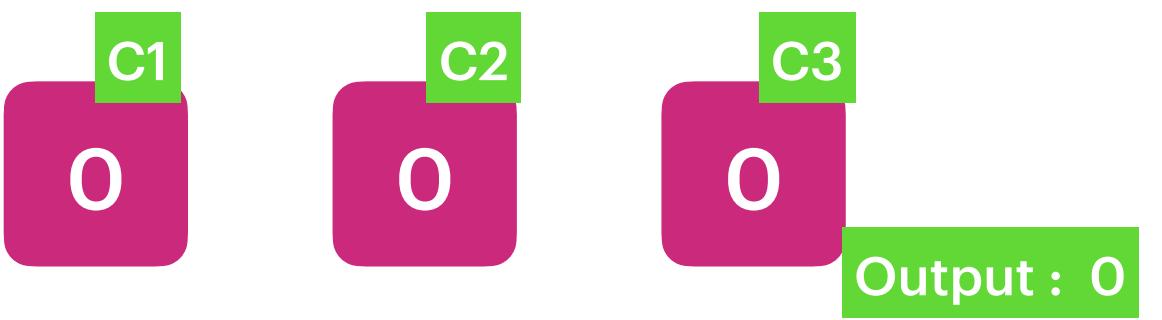
**Explanation:** Since there are already  
no fresh oranges at minute 0,  
the answer is just 0.

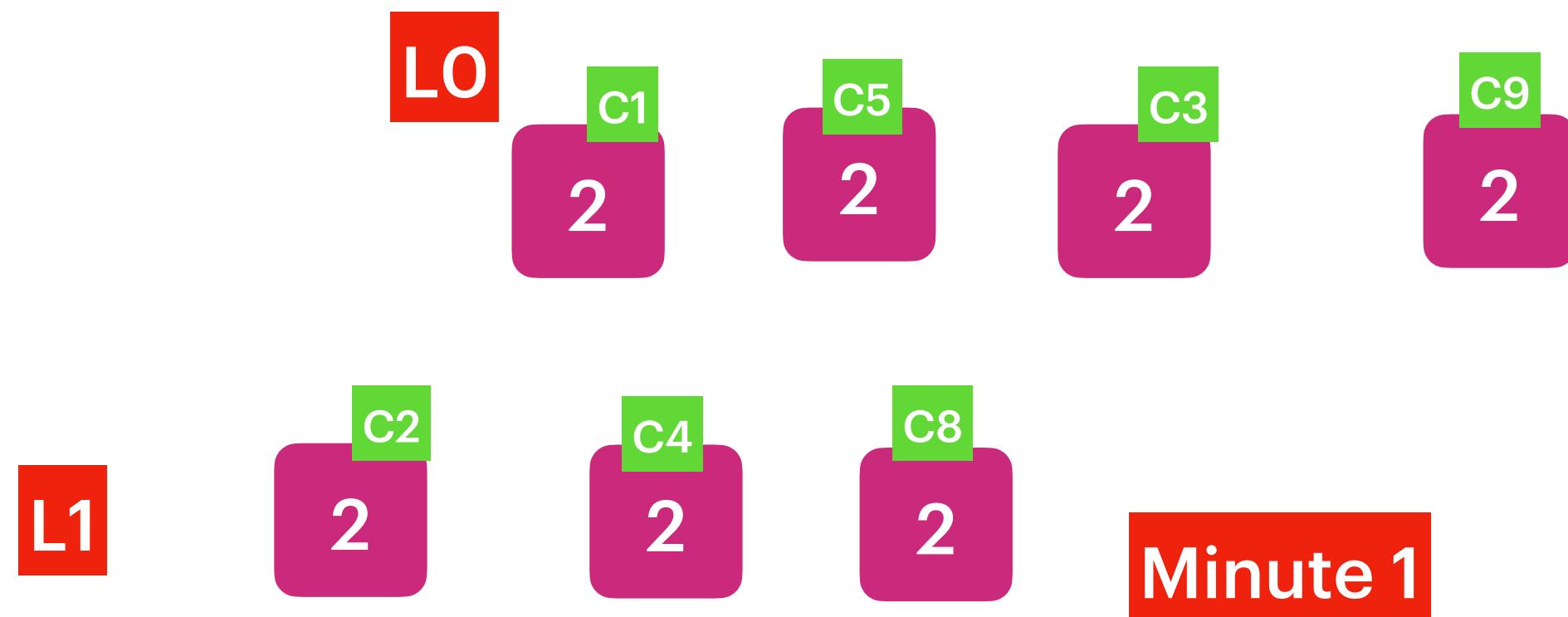
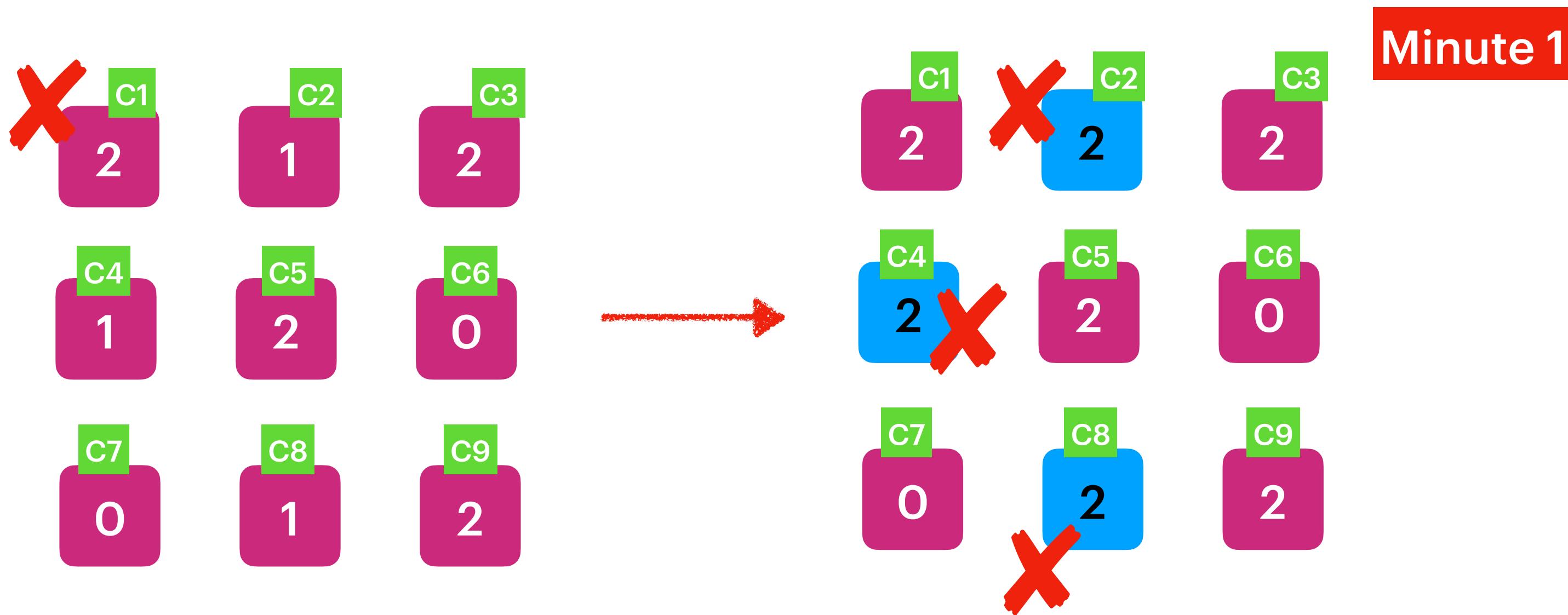
```
[  
[2,1,1],  
[1,1,0],  
[0,1,1]  
]
```











## Number of Islands

Given an  $m \times n$  2D binary grid grid which represents a map of '1's (land) and '0's (water), return the number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically.

You may assume all four edges of the grid are all surrounded by water.

**Input:** grid =  
["1","1","1","1","0"],  
["1","1","0","1","0"],  
["1","1","0","0","0"],  
["0","0","0","0","0"]  
]  
**Output:** 1

**Input:** grid =  
["1","1","0","0","0"],  
["1","1","0","0","0"],  
["0","0","1","0","0"],  
["0","0","0","1","1"]  
**Output:** 3

$m == \text{grid.length}$   
 $n == \text{grid[i].length}$   
 $1 \leq m, n \leq 300$   
 $\text{grid}[i][j]$  is '0' or '1'!

0	0	0	0	0	0
---	---	---	---	---	---

0	0	0	0	0	0
---	---	---	---	---	---

0	0	0	0	0	0
---	---	---	---	---	---

islandsCount : 0

0	0	0	0	0	0
---	---	---	---	---	---

0	0	0	1	0	0
---	---	---	---	---	---

islandsCount : 1

0	0	0	0	0	0
---	---	---	---	---	---

0	0	1	0	0	0
---	---	---	---	---	---

0	0	0	1	0	0
---	---	---	---	---	---

islandsCount : 3

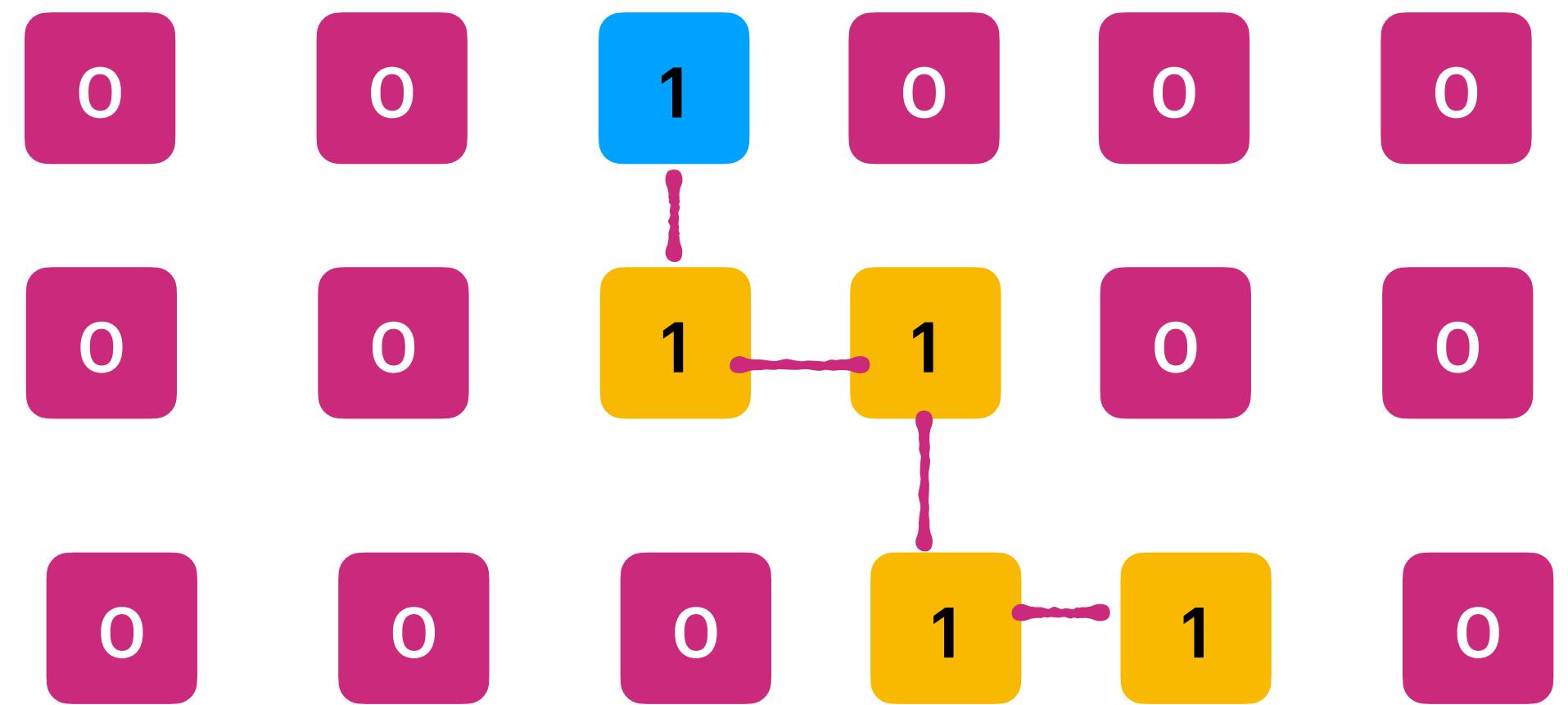
0	0	0	0	1	0
---	---	---	---	---	---

0	0	1	0	0	0
---	---	---	---	---	---

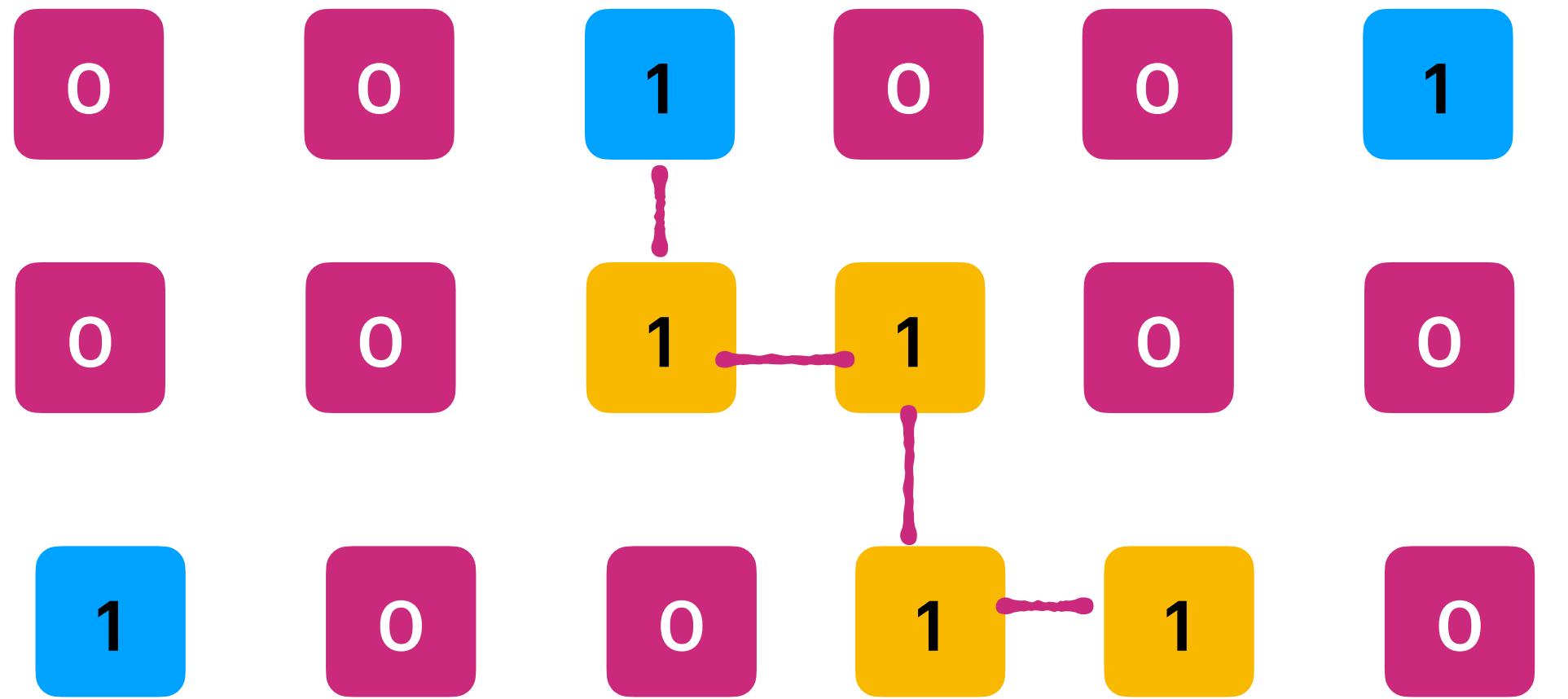
0	0	1	0	0	0
---	---	---	---	---	---

islandsCount : 2

0	0	0	0	1	0
---	---	---	---	---	---



islandsCount : 1



islandsCount : 3

0	0	1	0	0	0
0	0	1	1	0	0
0	0	0	0	1	0

Total Islands : 2

0	0	1	1	0	0
0	0	0	0	1	0
0	0	0	0	0	0

```

for ( r : Row )
{
    for(r : Col)
    {
        if(cell[r][c] = bfs[r,c];
        islandCount+
    }
}

```

0	0	1	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Step1

BFS/  
DFS

islandCount = 1

0	0	1	0	0	0
0	0	0	0	0	0
0	0	0	0	1	0

Step2

islandCount = 2

## Shortest Path in Binary Matrix

Given an  $n \times n$  binary matrix grid, return the length of the shortest clear path in the matrix.

If there is no clear path, return -1.

A clear path in a binary matrix is a path from the top-left cell (i.e.,  $(0, 0)$ ) to the bottom-right cell (i.e.,  $(n - 1, n - 1)$ ) such that:

All the visited cells of the path are 0.

All the adjacent cells of the path are 8-directionally connected (i.e., they are different and they share an edge or a corner).

The length of a clear path is the number of visited cells of this path.

**Input:** grid = [[0,1],[1,0]]

**Output:** 2

**Input:** grid = [[0,0,0],[1,1,0],[1,1,0]]

**Output:** 4

**Constraints:**

$n == \text{grid.length}$

$n == \text{grid[i].length}$

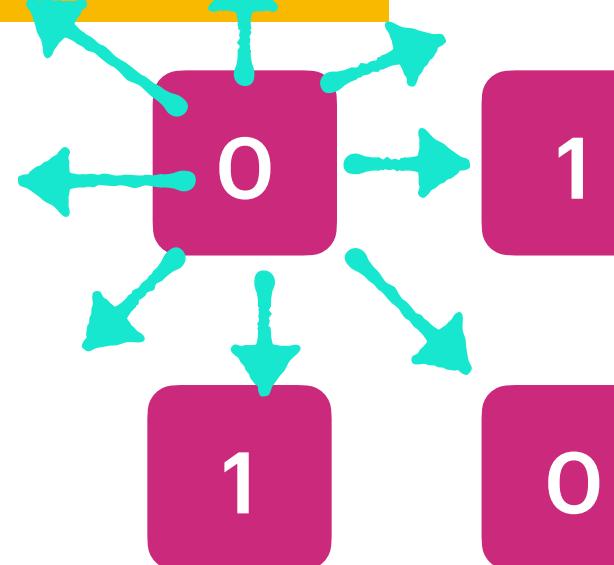
$1 \leq n \leq 100$

$\text{grid[i][j]}$  is 0 or 1

**Input:** grid = [[1,0,0],[1,1,0],[1,1,0]]

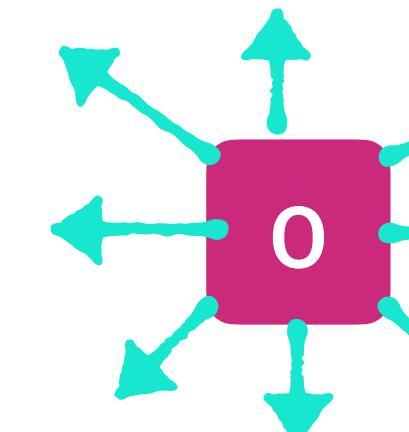
**Output:** -1

**Input:** grid = [[0,1],[1,0]]  
**Output:** 2



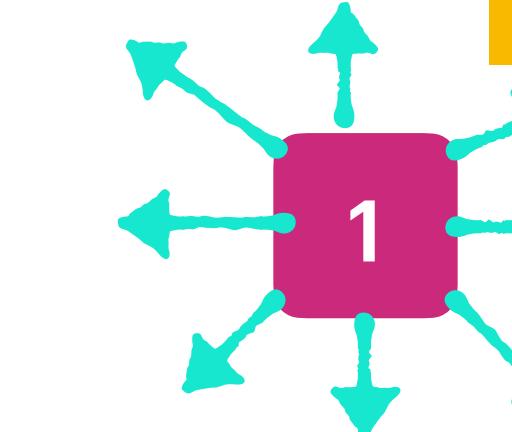
**Output :** 2

**Input:** grid = [[0]]  
**Output:** 1



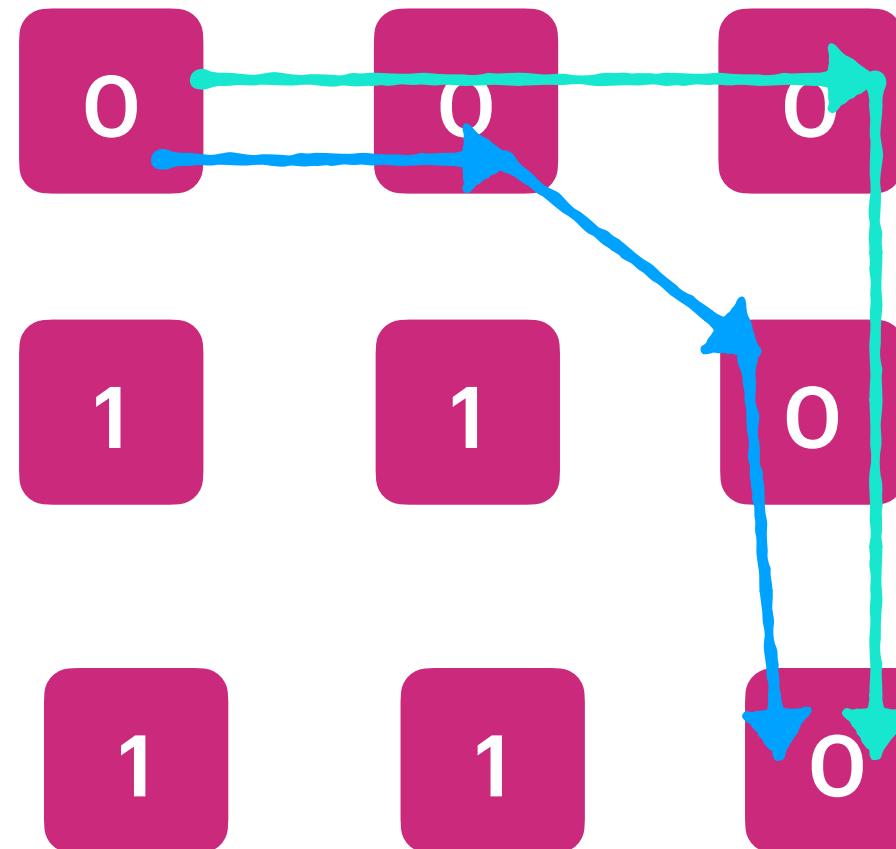
**Output :** 1

**Input:** grid = [[1]]  
**Output:** -1



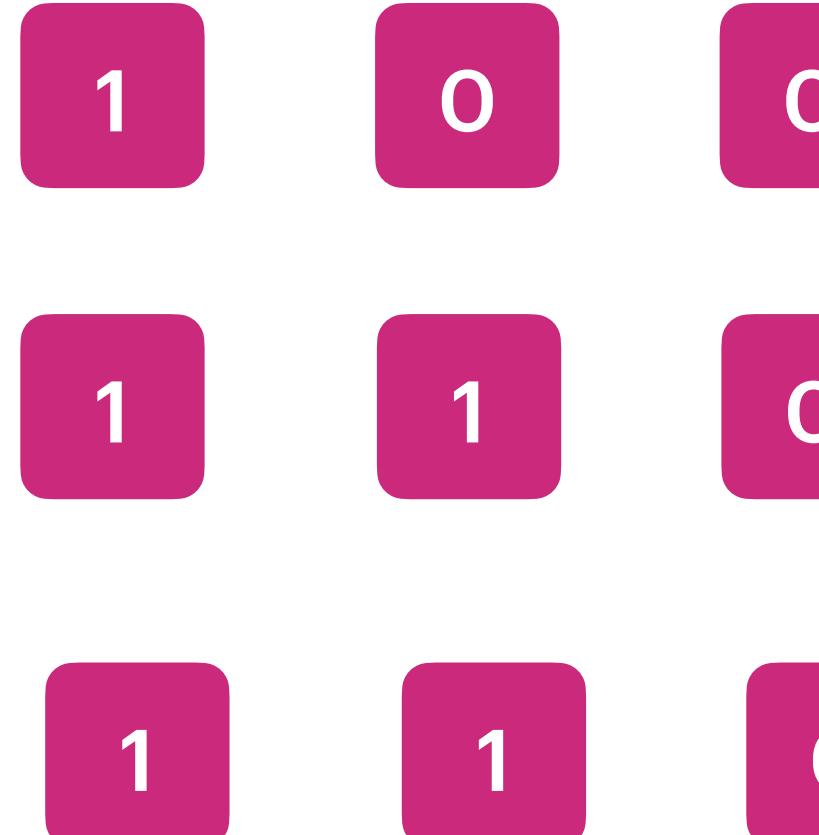
**Output :** -1

**Input:** grid = [[0,0,0],[1,1,0],[1,1,0]]  
**Output:** 4



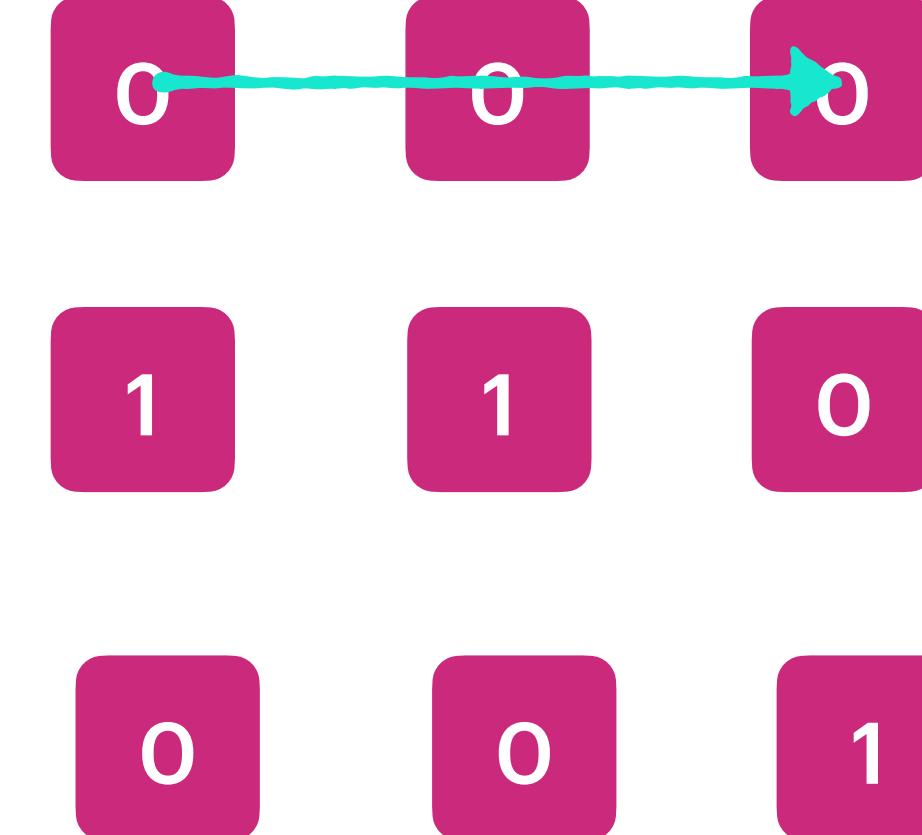
**Output :** 4

**Input:** grid = [[1,0,0],[1,1,0],[1,1,0]]  
**Output:** -1

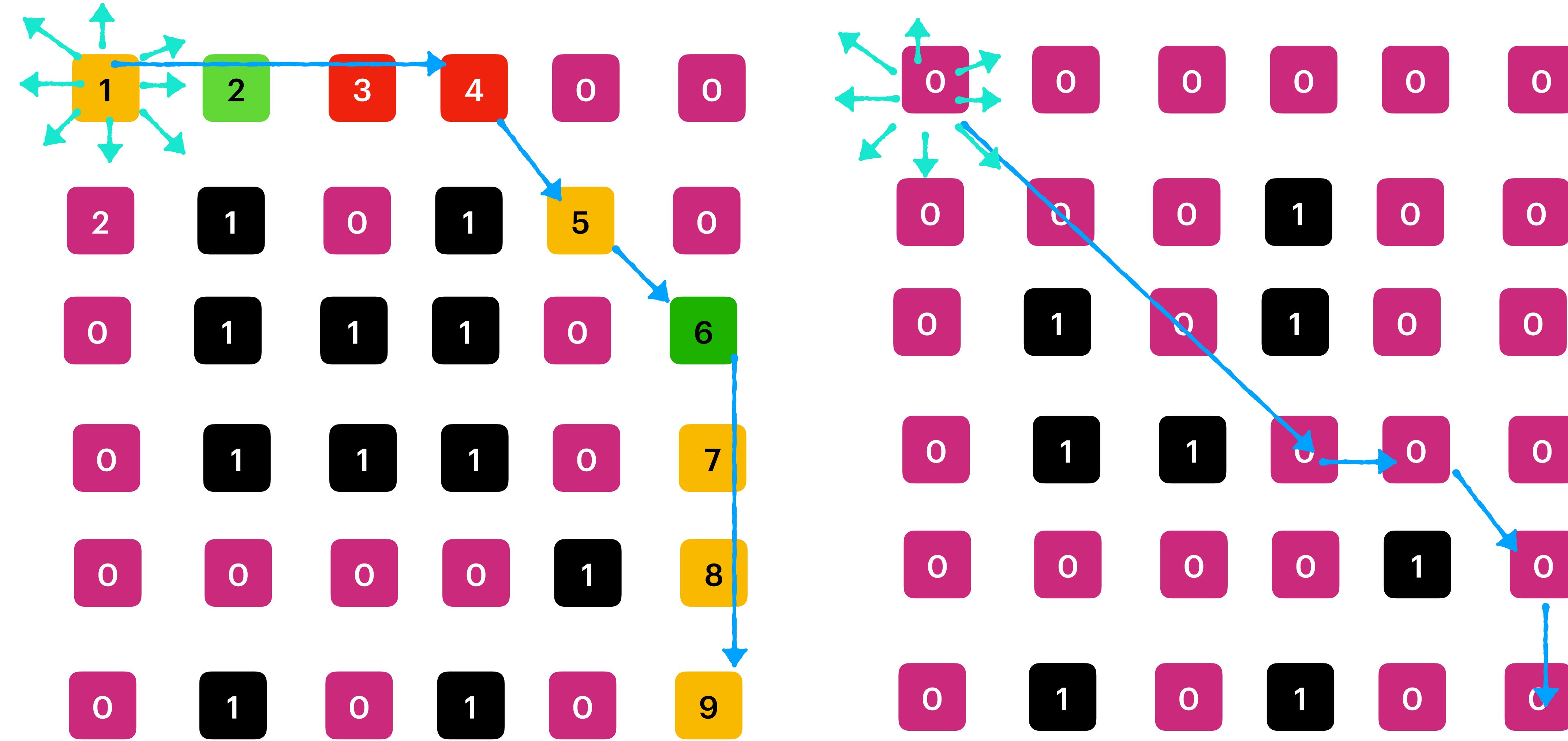


**Output :** -1

**Input:** grid = [[0,0,0],[1,1,0],[0,0,0]]  
**Output:** -1

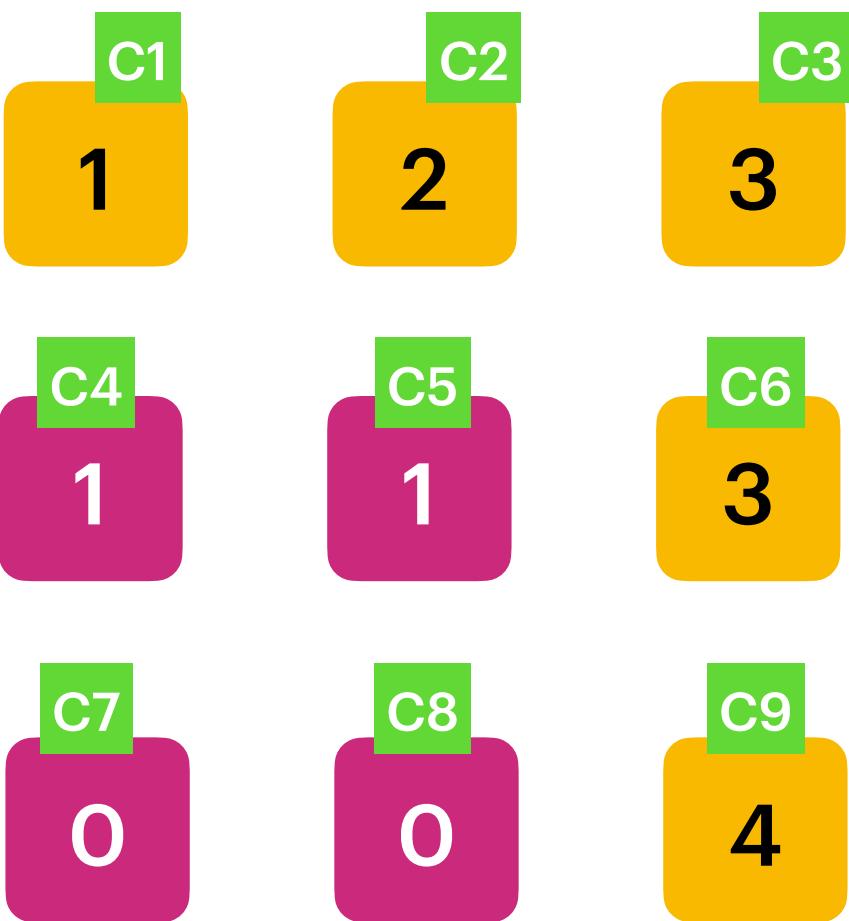


**Output :** -1

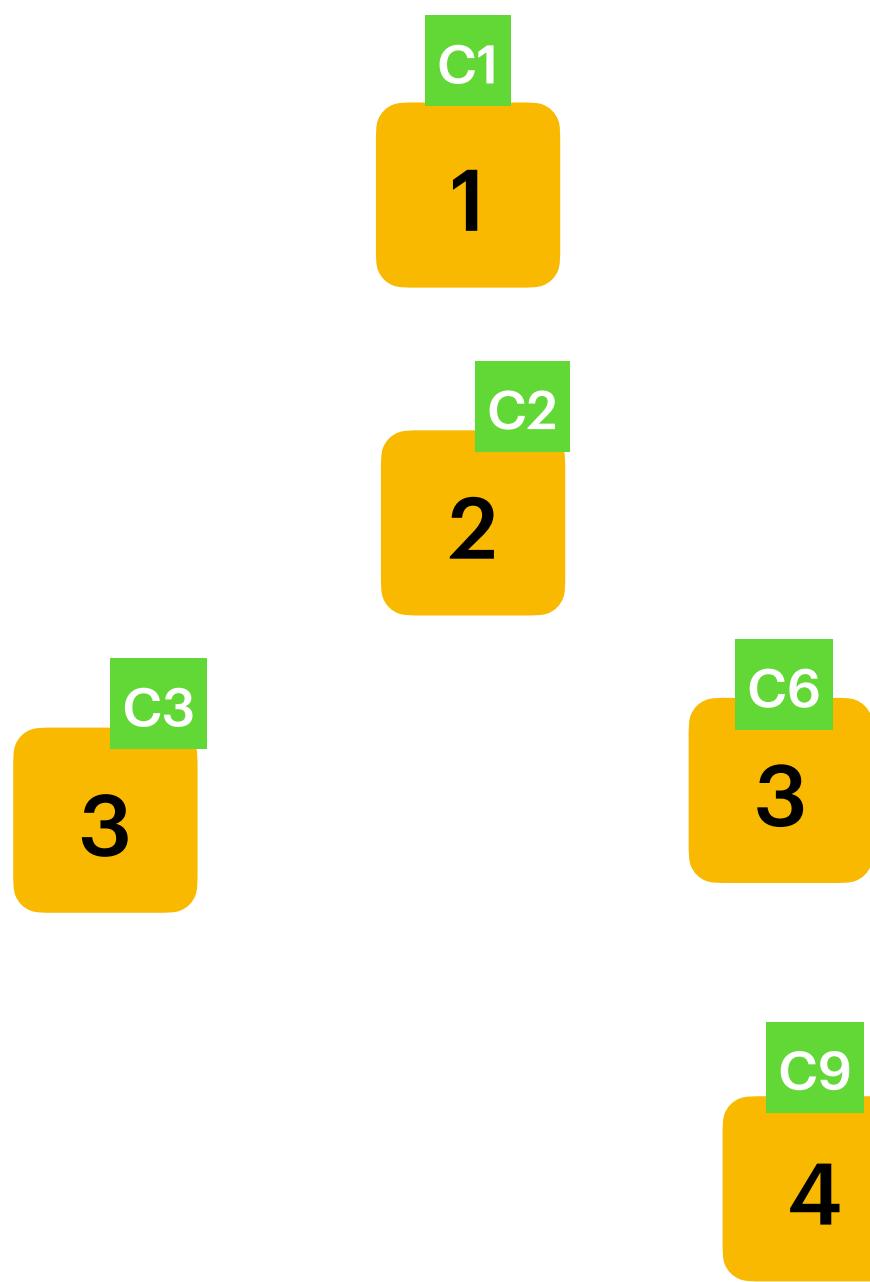


Output : 9

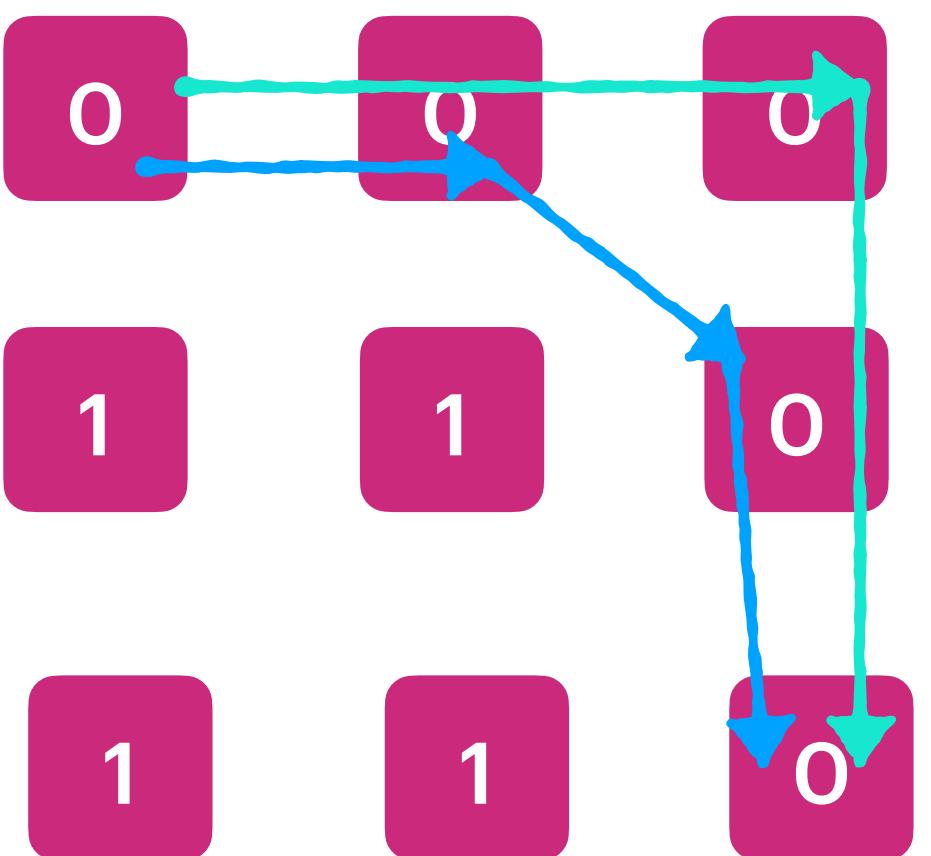
Output : 7



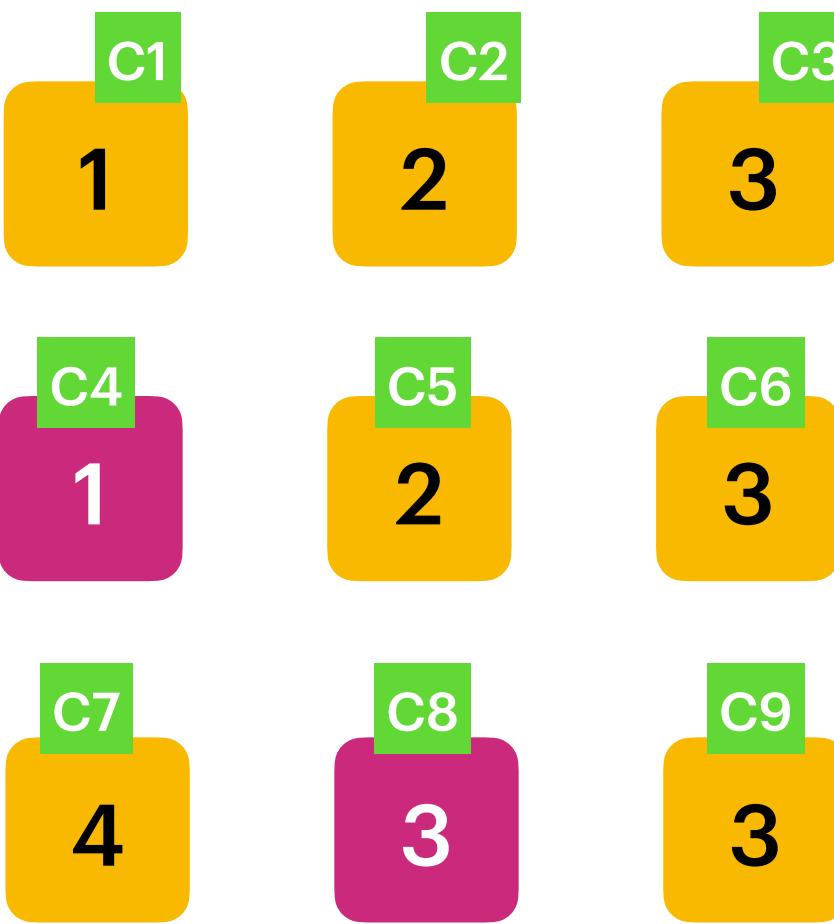
=



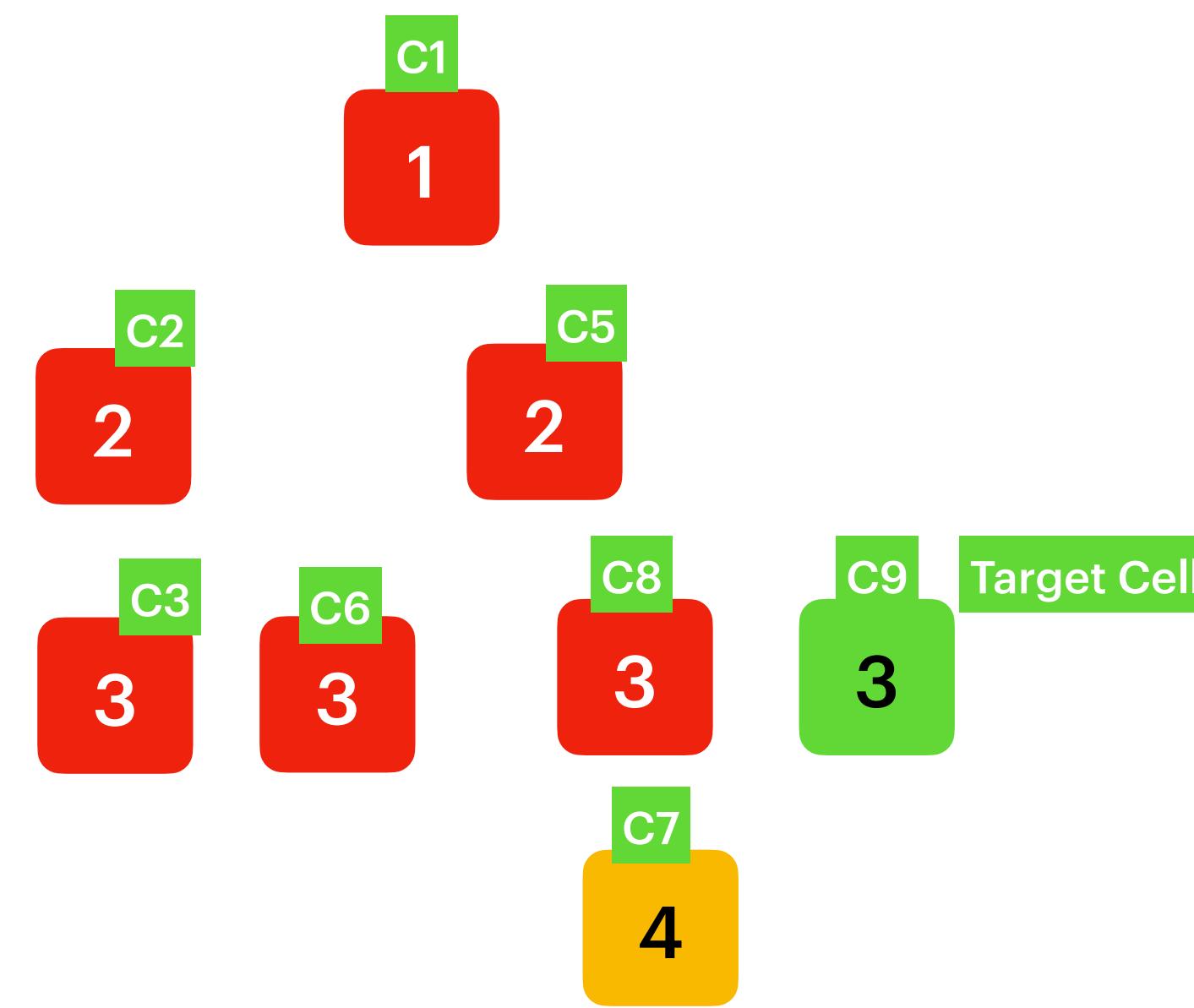
**Input:** grid = [[0,0,0],[1,1,0],[1,1,0]]  
**Output:** 4



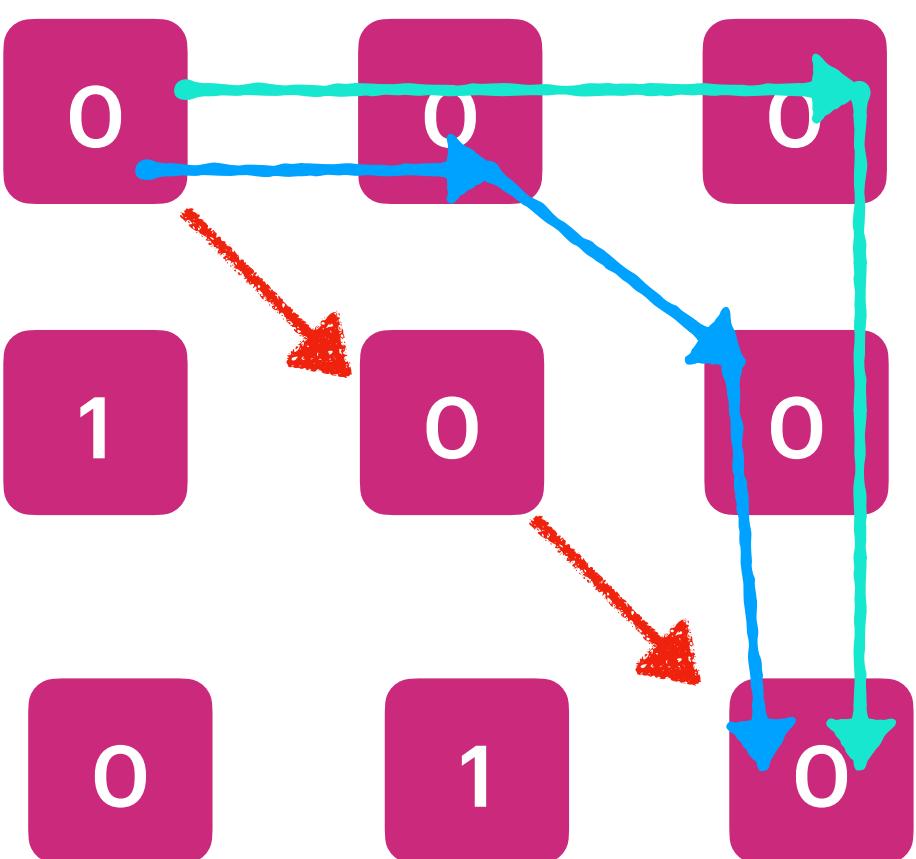
**Output :** 4



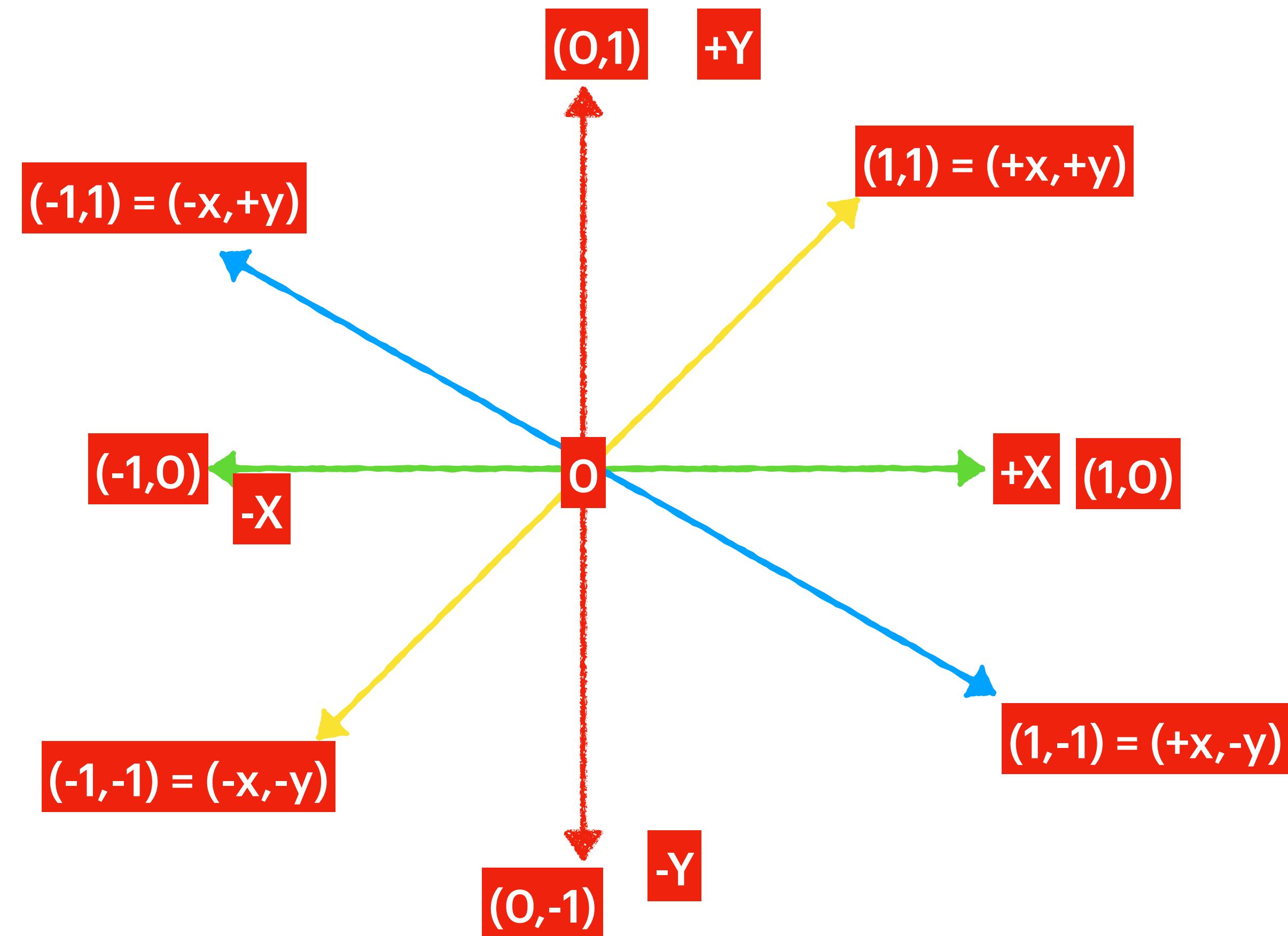
=



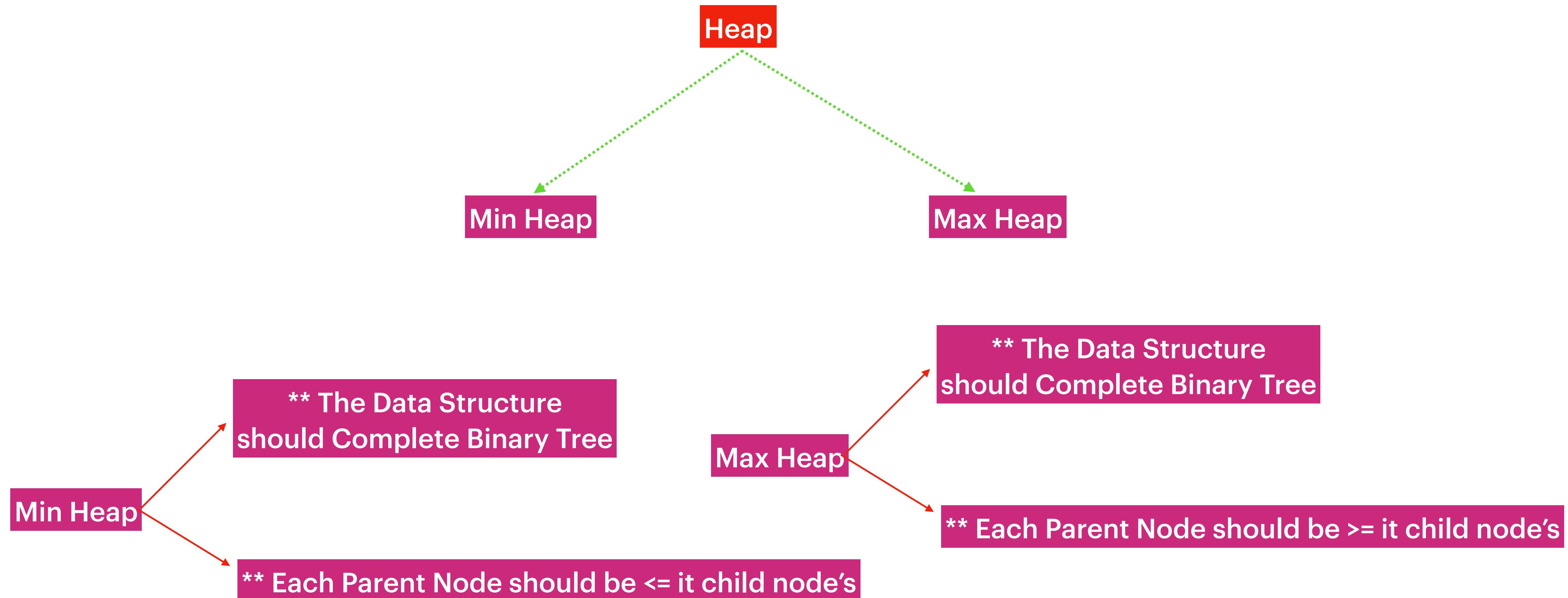
**Input:** grid = [[0,0,0],[1,1,0],[1,1,0]]  
**Output:** 4



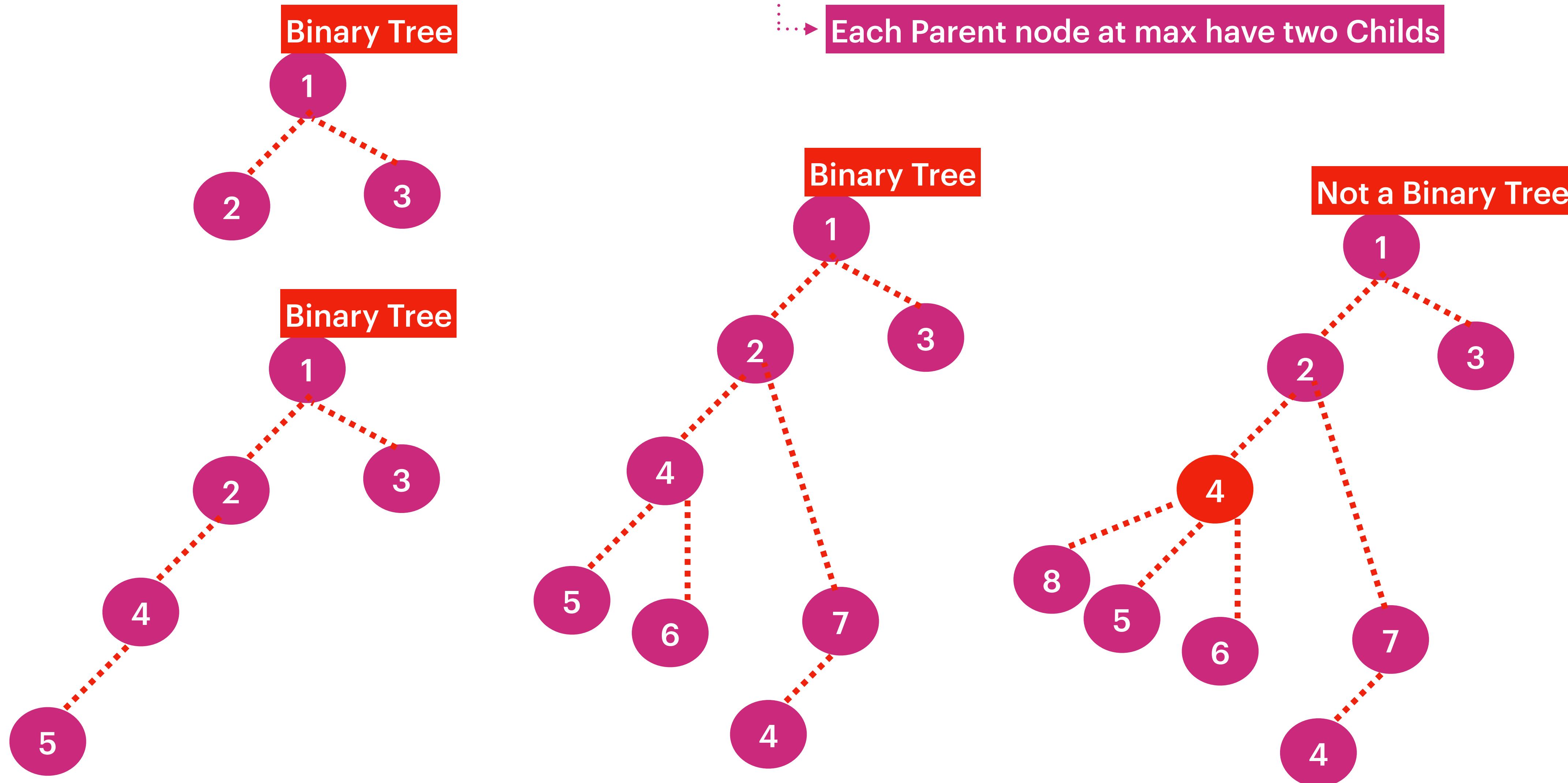
**Output :** 4



## Let's Understand Heap Before moving on to weighted Graphs

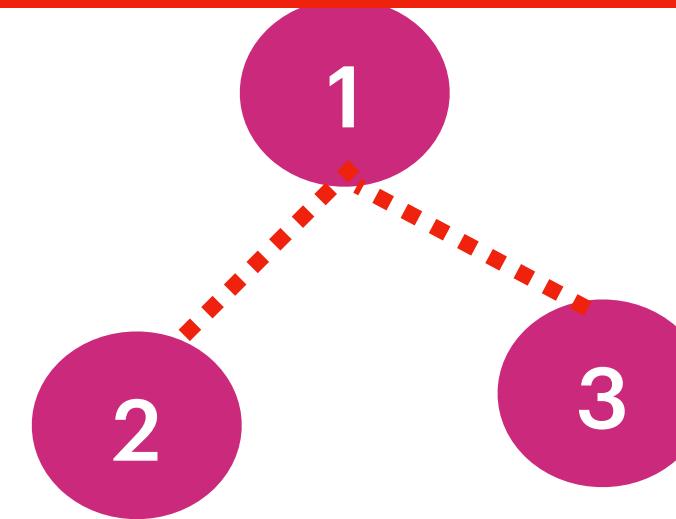


What is Binary Tree ?



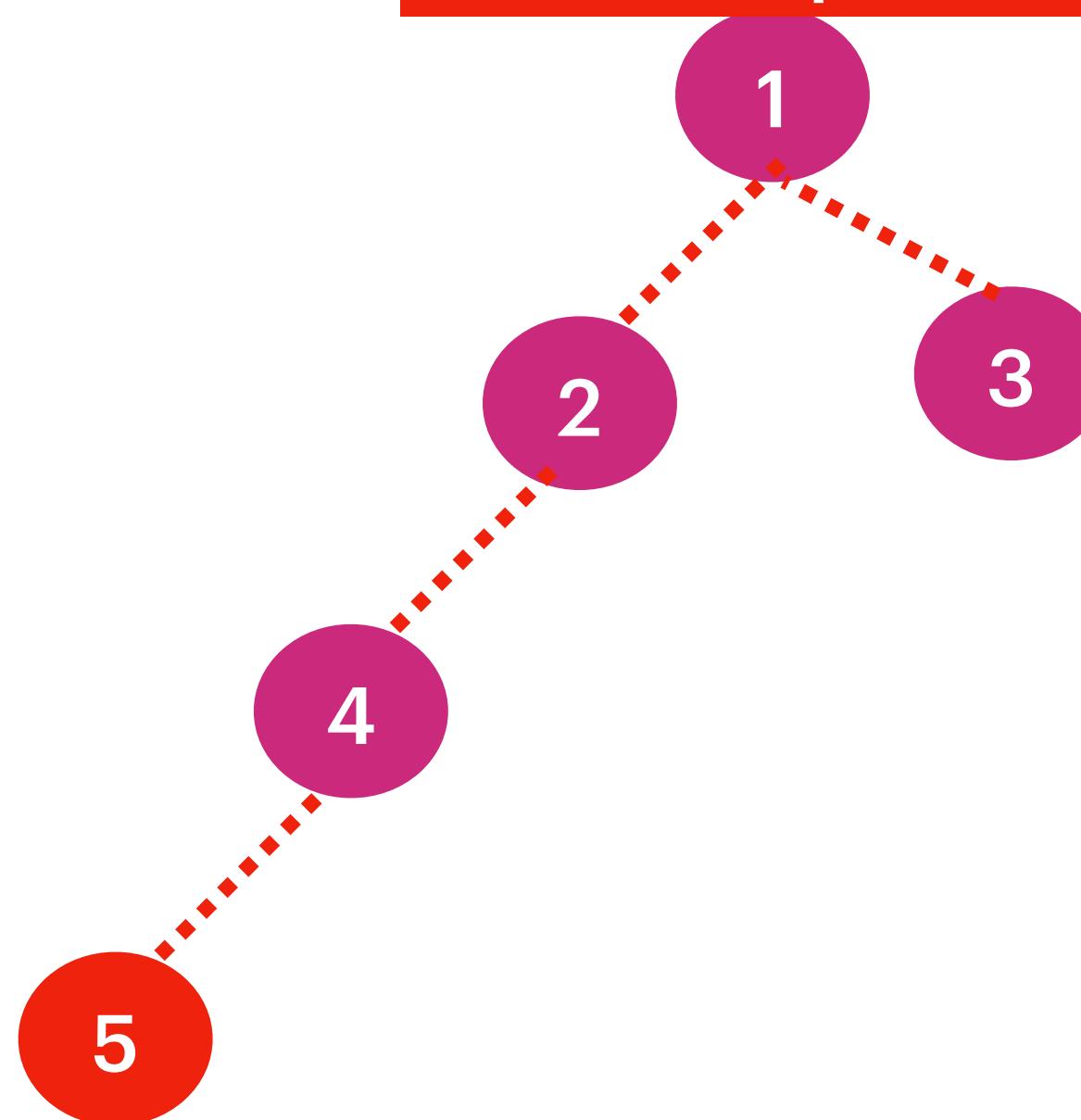
## What is Complete Binary Tree ?

Complete Binary Tree

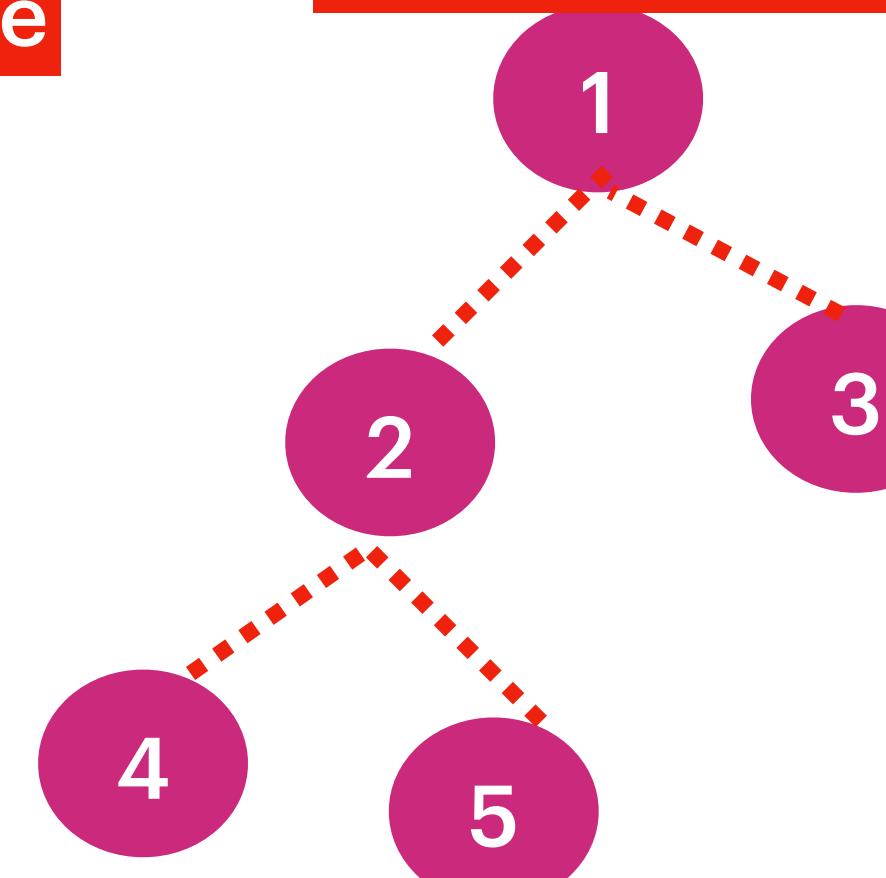


Each Parent node at max have two Childs,  
At Each level order of elements should  
Be filled from left -> right.  
Once the current level complete we can  
Move on to next level.

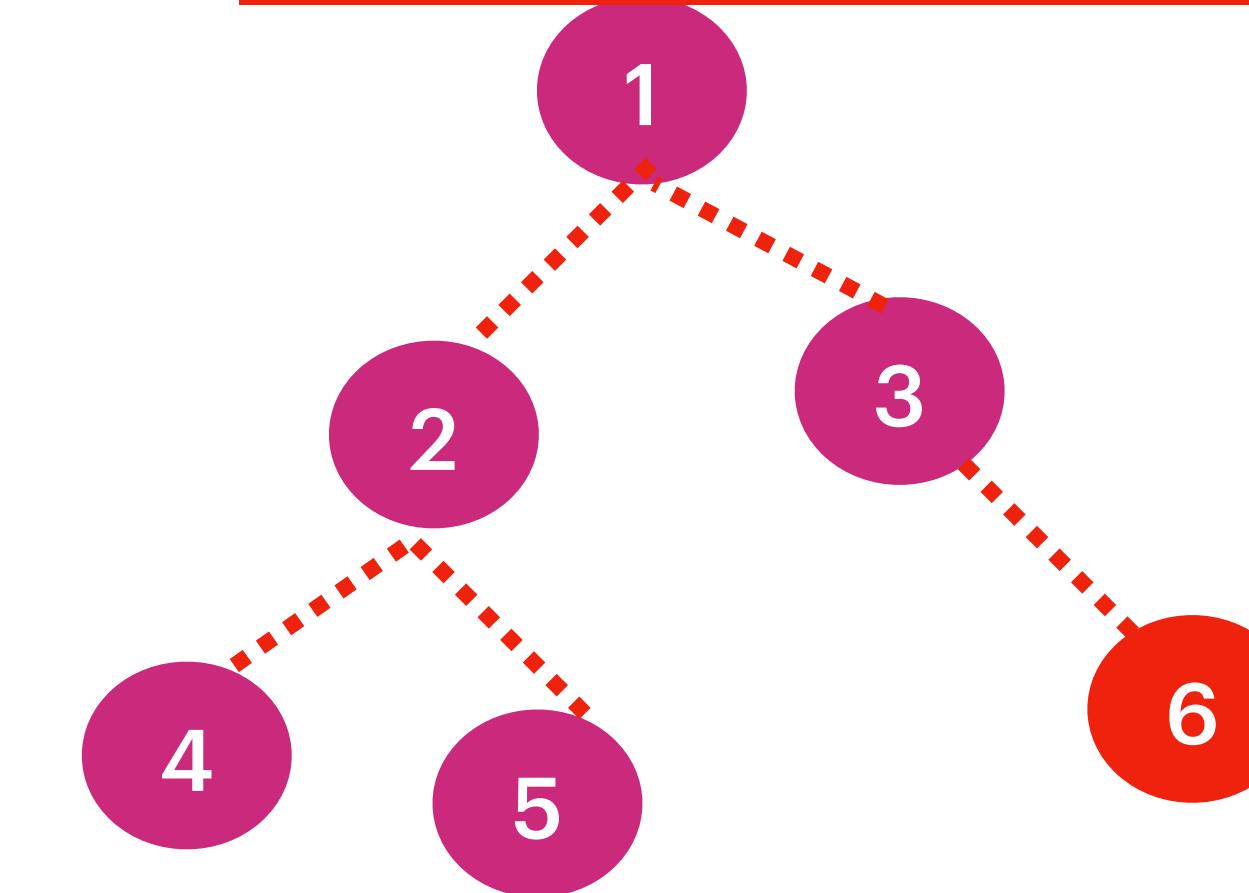
Not a Complete Binary Tree



Complete Binary Tree

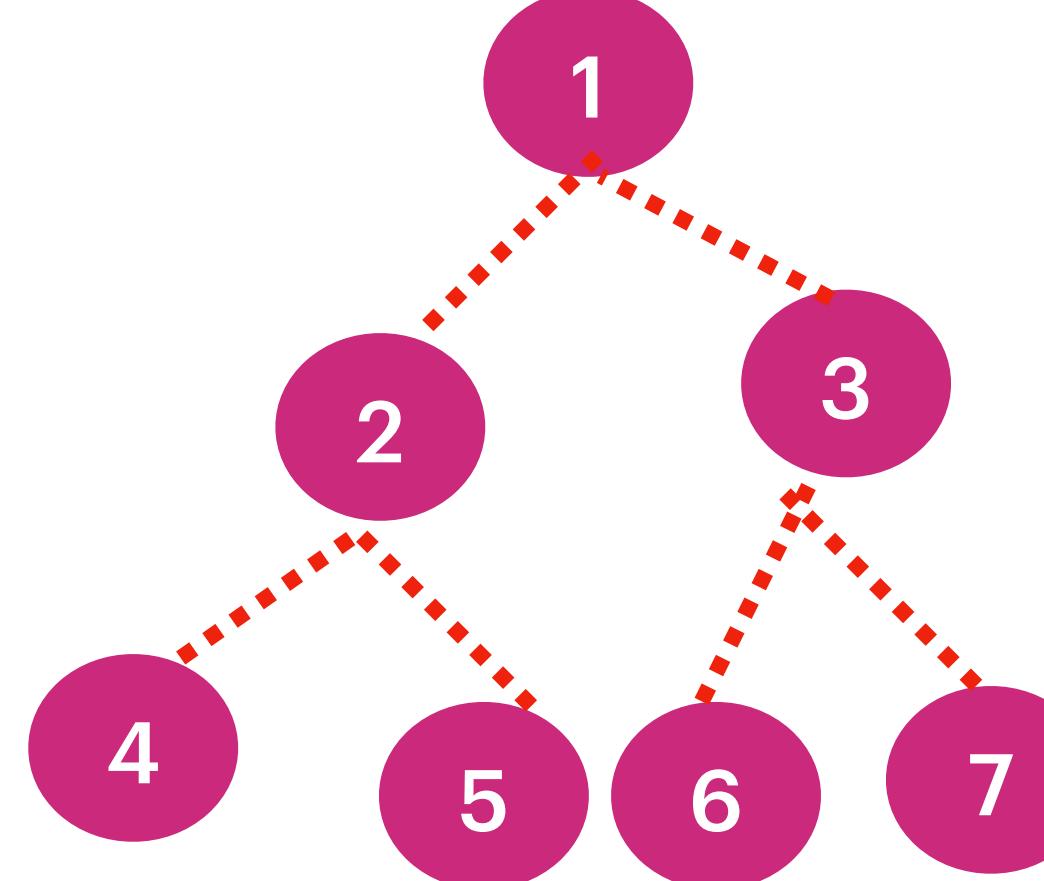


Not a Complete Binary Tree



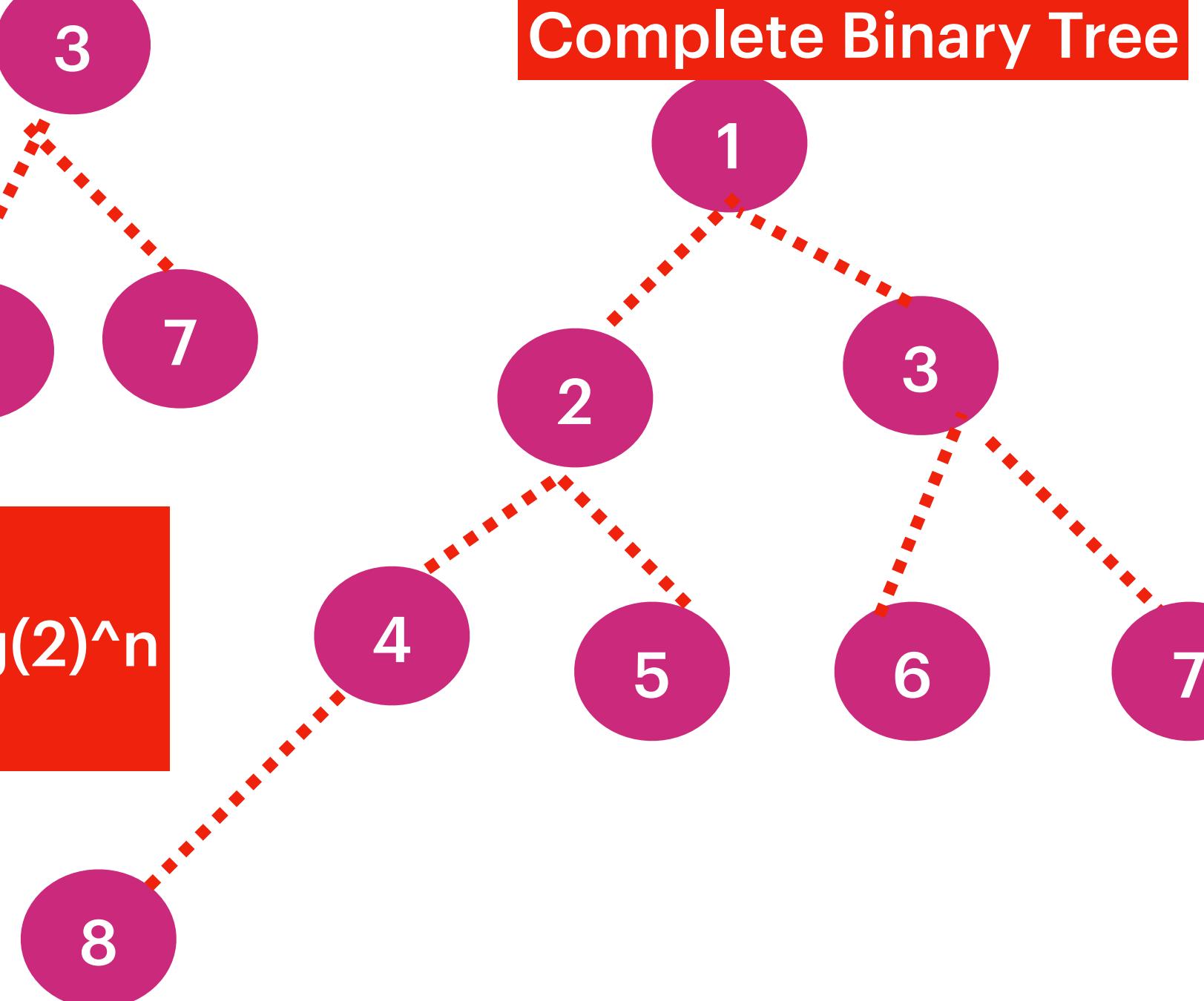
## What is Complete Binary Tree ?

Complete Binary Tree

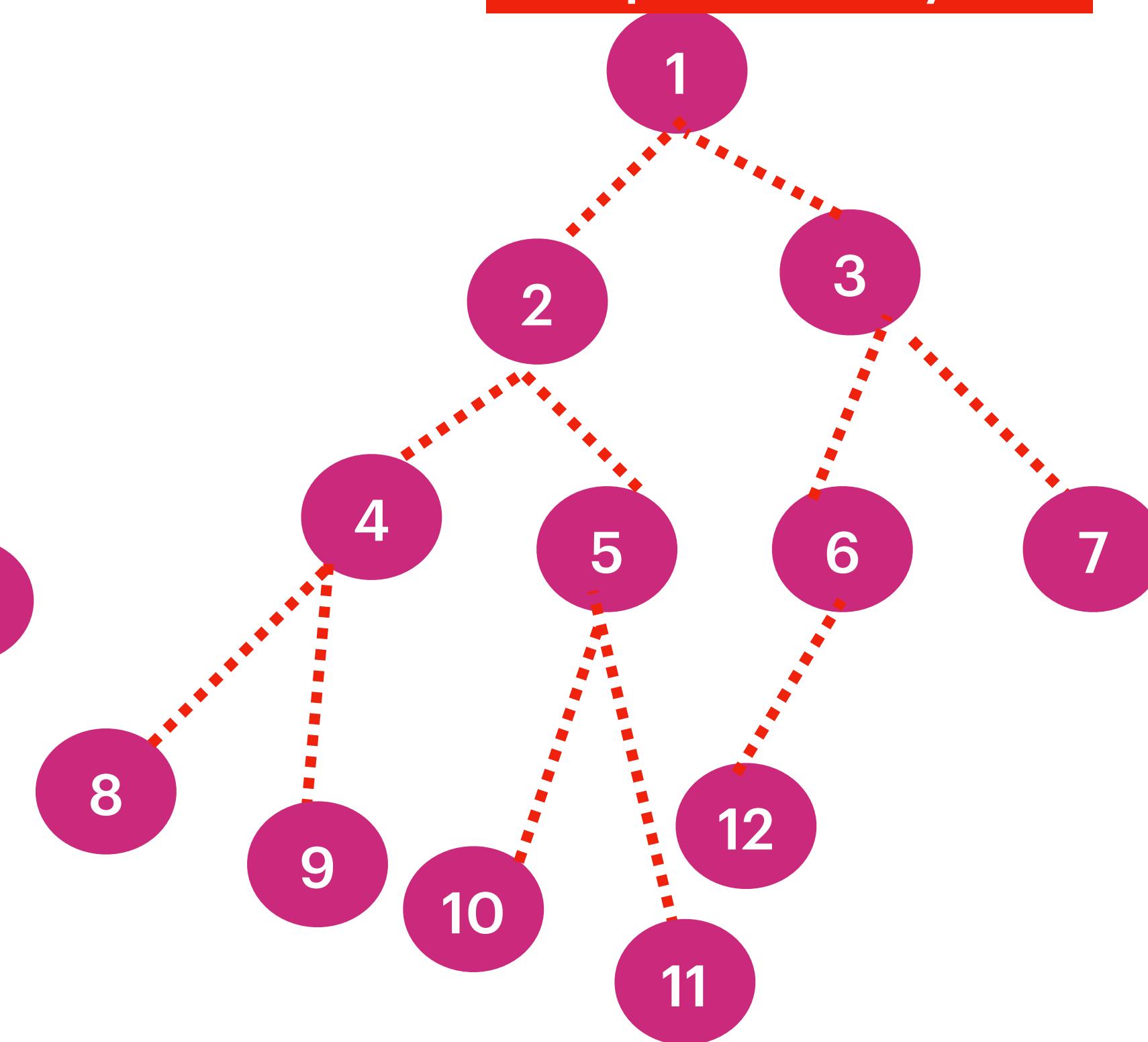


Each Parent node at max have two Childs,  
At Each level order of elements should  
Be filled from left -> right.  
Once the current level complete we can  
Move on to next level.

Complete Binary Tree

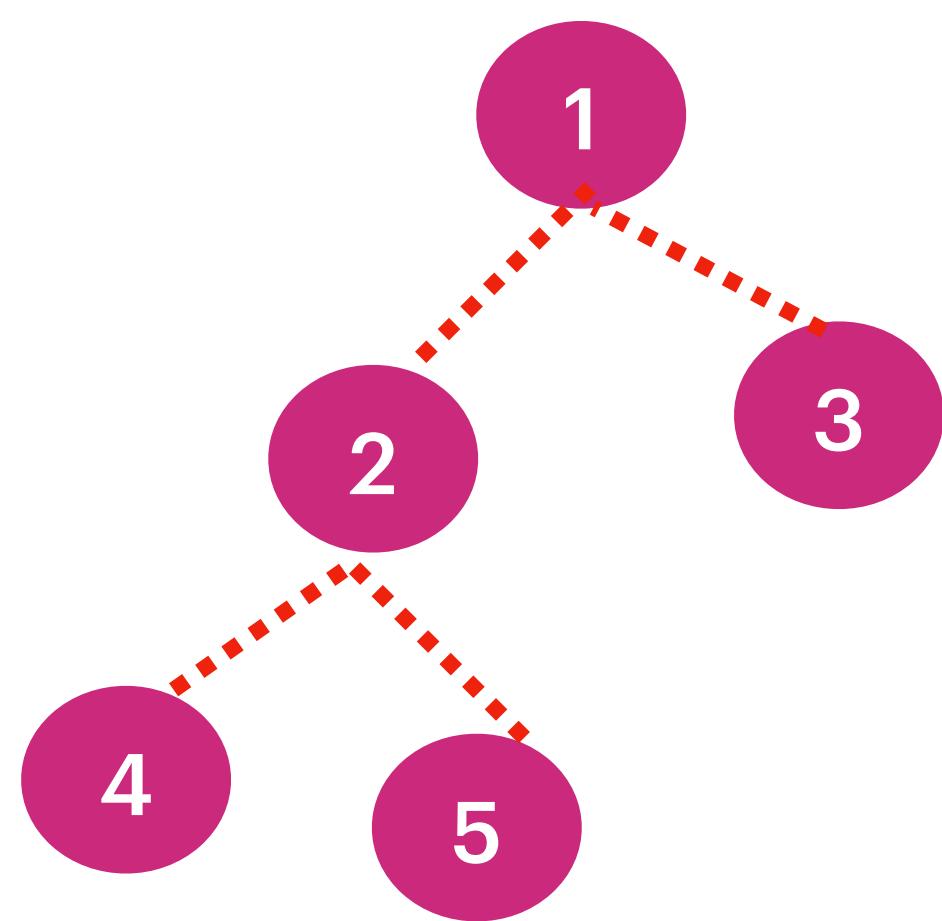


Complete Binary Tree



Height  
Of Complete Binary Tree :  $\log(2)^n$

## Complete Binary Tree



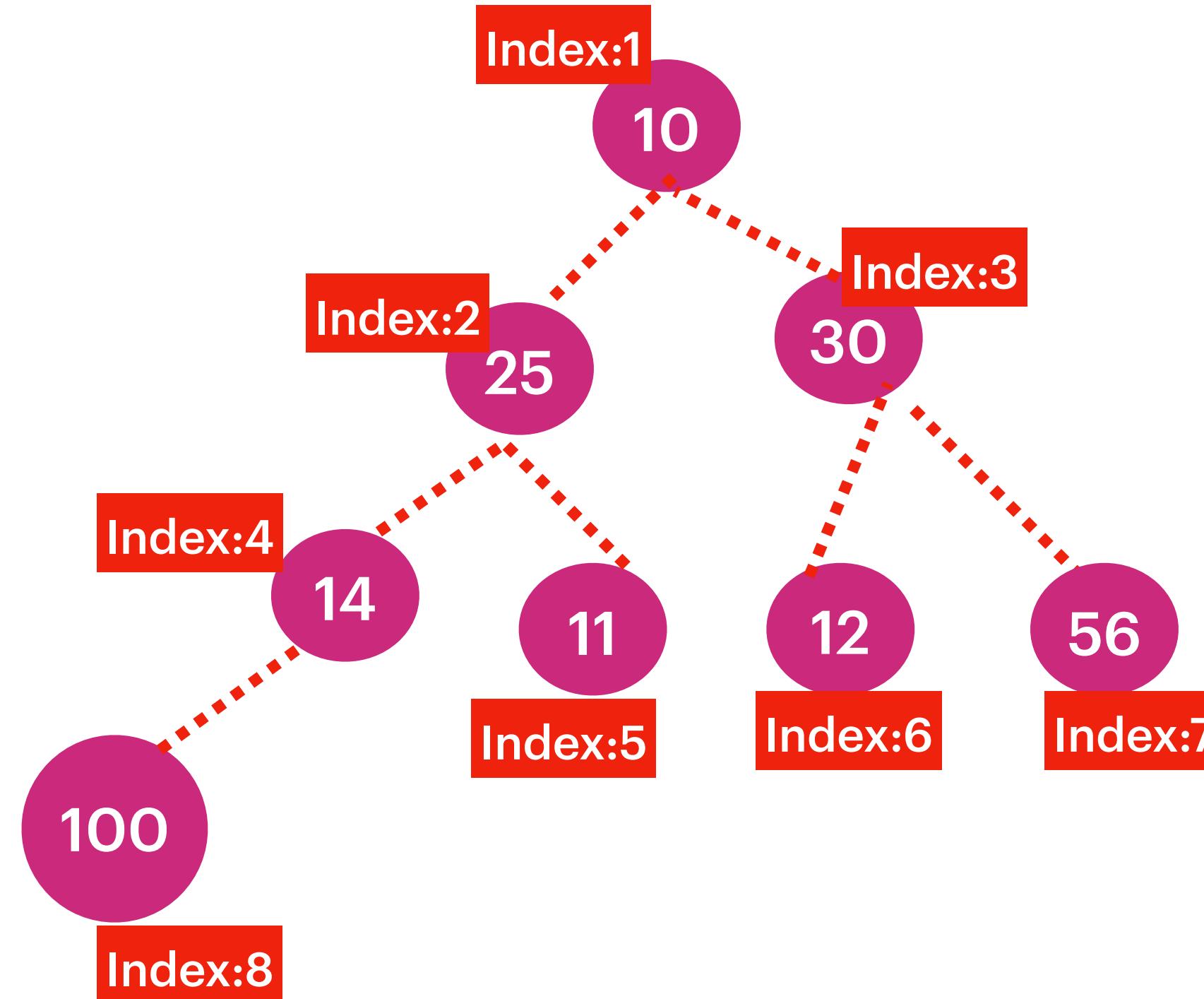
Root : 1

Leaf Nodes : 3, 4, 5

A Node which does not have Childs is a leafNode

Height : 3 ~  $\log(2)^n$

## Complete Binary Tree



**Parent = index/2;**

**Ex: Parent Of 25 =  
2/2 = index:1 = 10**

## Transform Complete Binary Tree to the Hash/Array

**How do you find parent ?**

**How do you find left & right child ?**

**How do you know that current element is a leaf Node ?**

**Left Child = index \* 2**

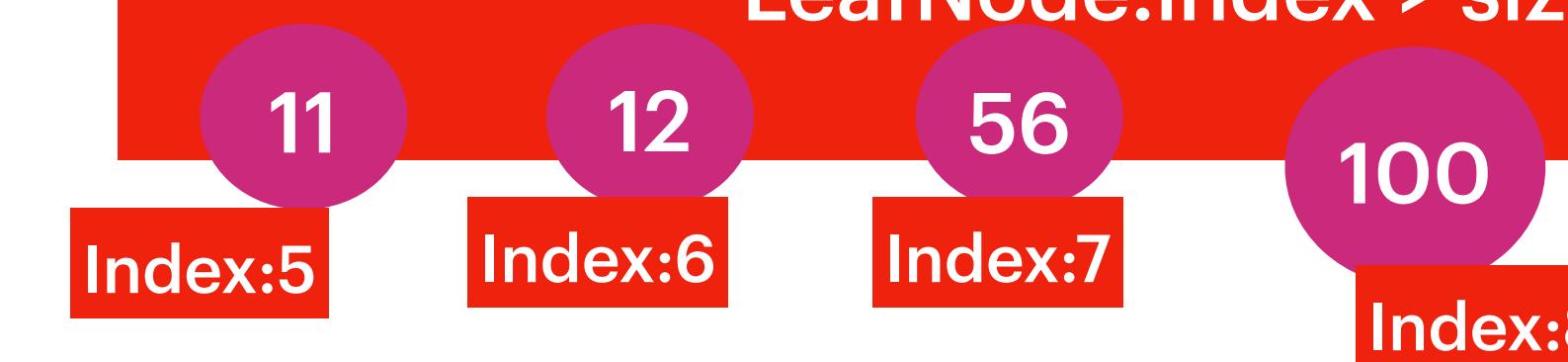
**Right Child = index \* 2 + 1**

**Ex: Left Child of Element 30 =  $2 * \text{index:3} = \text{index:6} = 12$**

**Right Child of Element 30 =  $2 * \text{index:3} + 1 = \text{index:7} = 56$**

**How do you know that current element is a leaf Node ?**

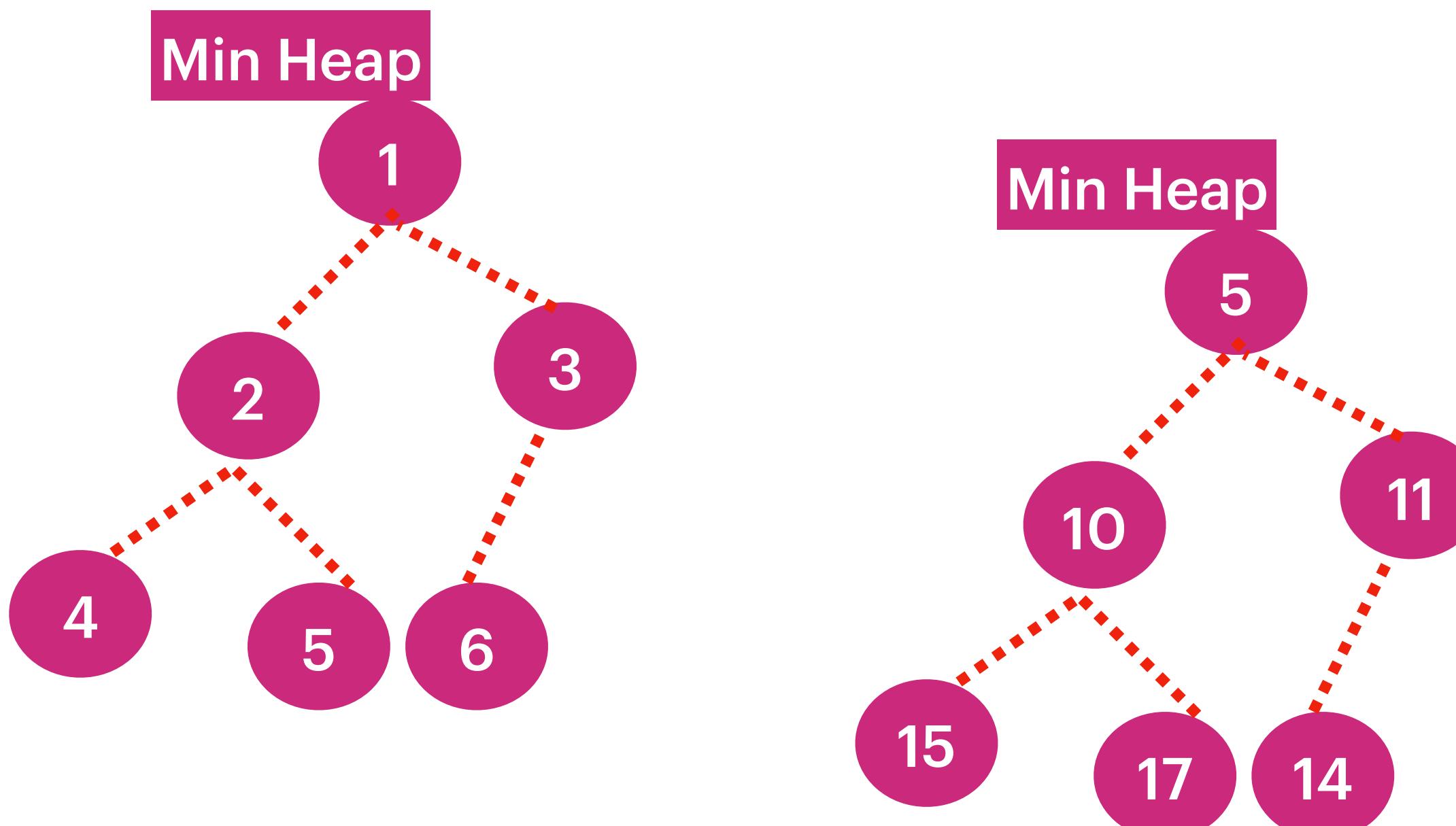
**LeafNode:index > size/2**



**\*\* The Data Structure  
should Complete Binary Tree**

**Min Heap**

**\*\* Each Parent Node should be <= it child node's**



## Insert Operation On MinHeap

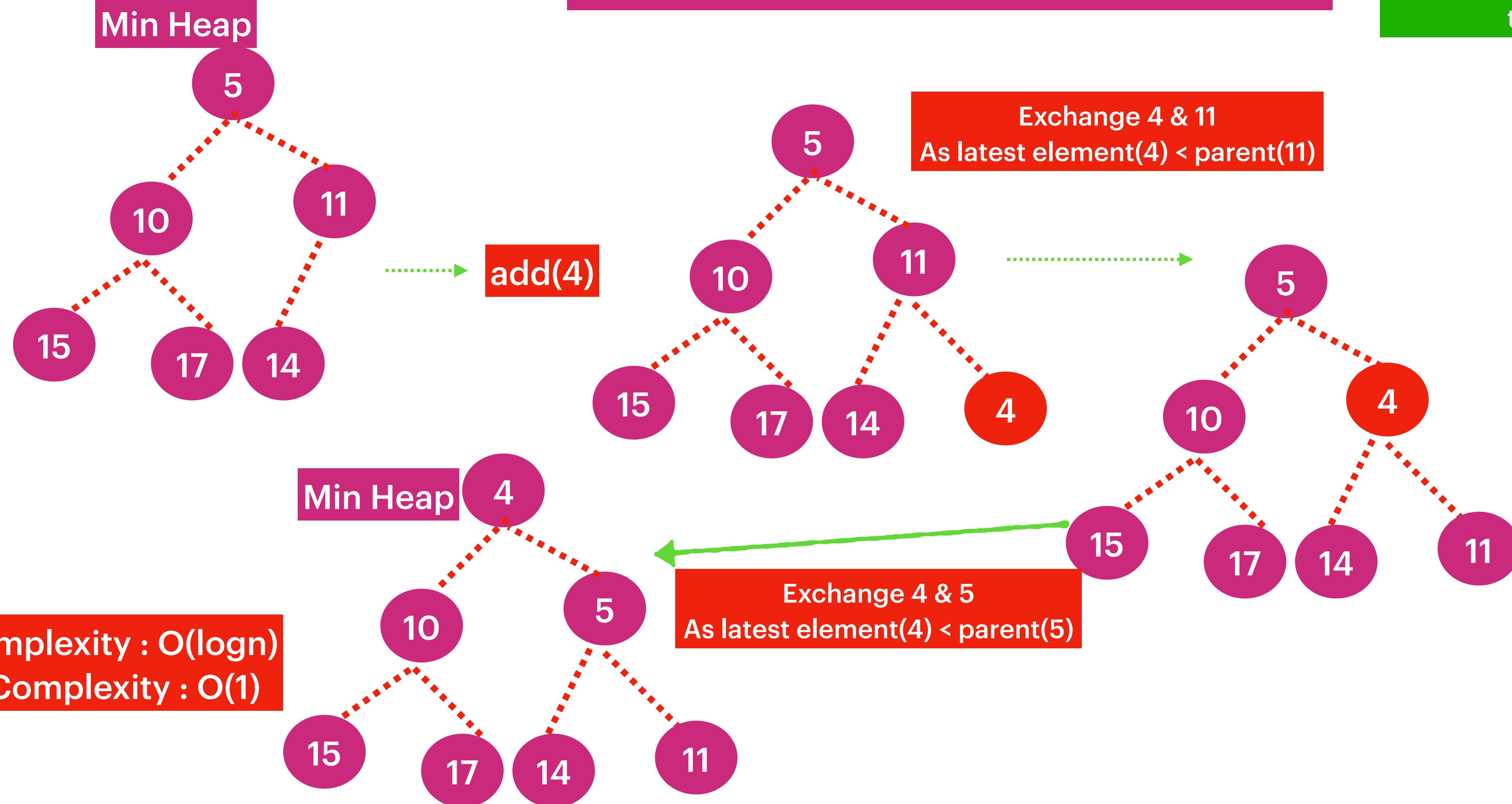
\*\* The Data Structure  
should Complete Binary Tree

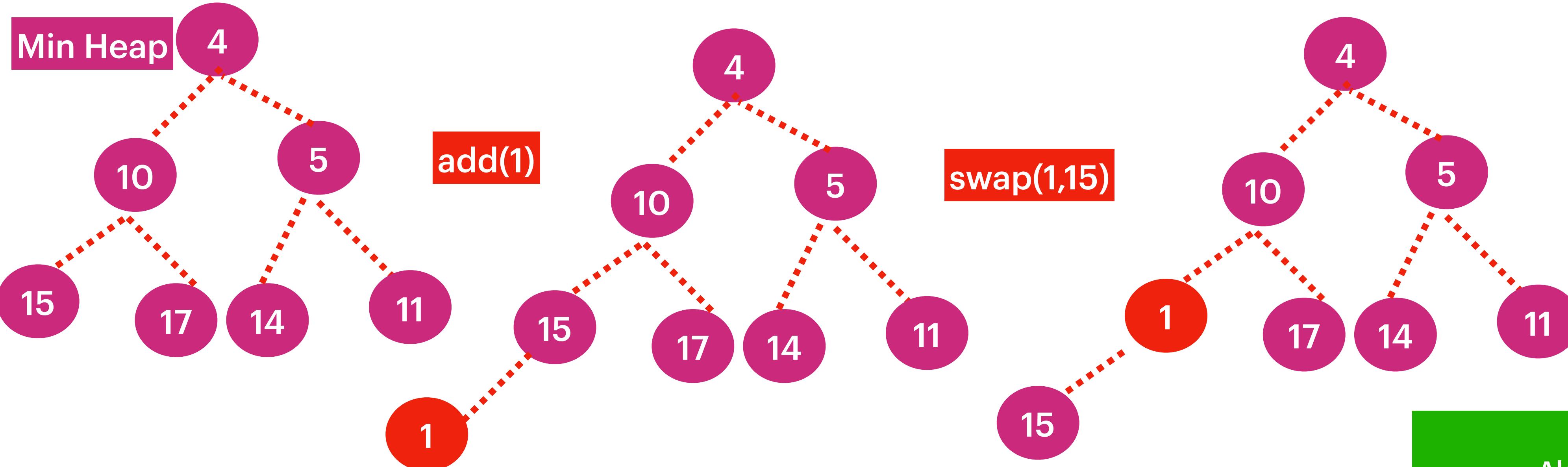
Min Heap

\*\* Each Parent Node should be  $\leq$  it child node's

Algorithm :

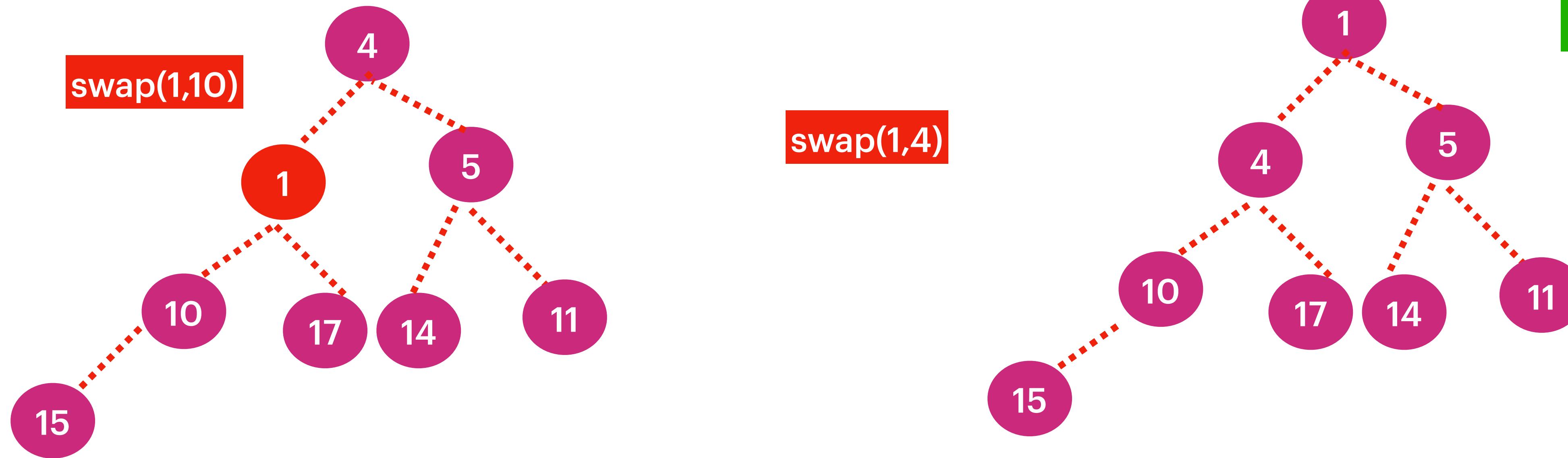
Add to the RightMost position.  
If the current element > parent  
then swap.



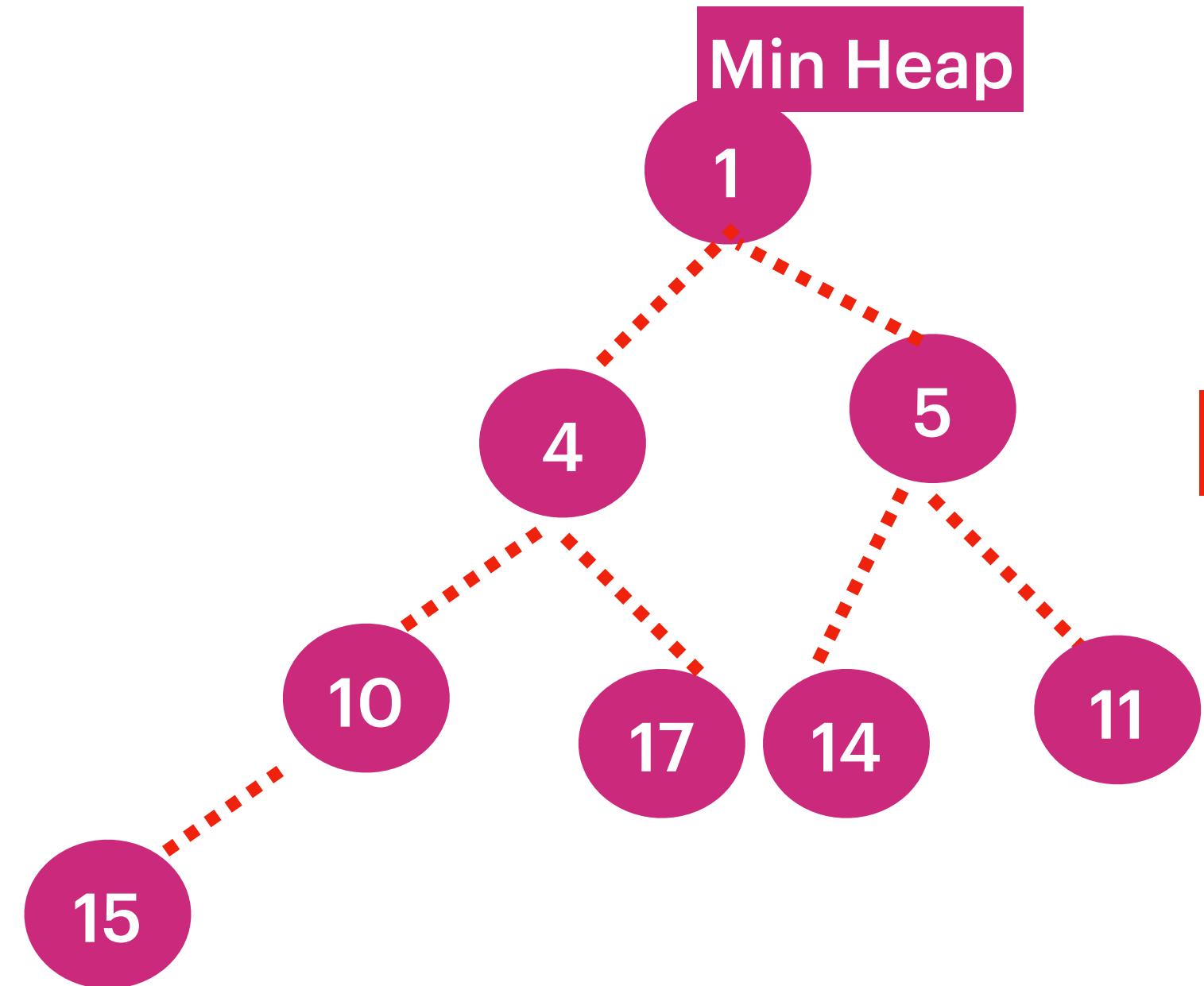


**Algorithm :**

Add to the RightMost position.  
If the current element > parent  
then swap.



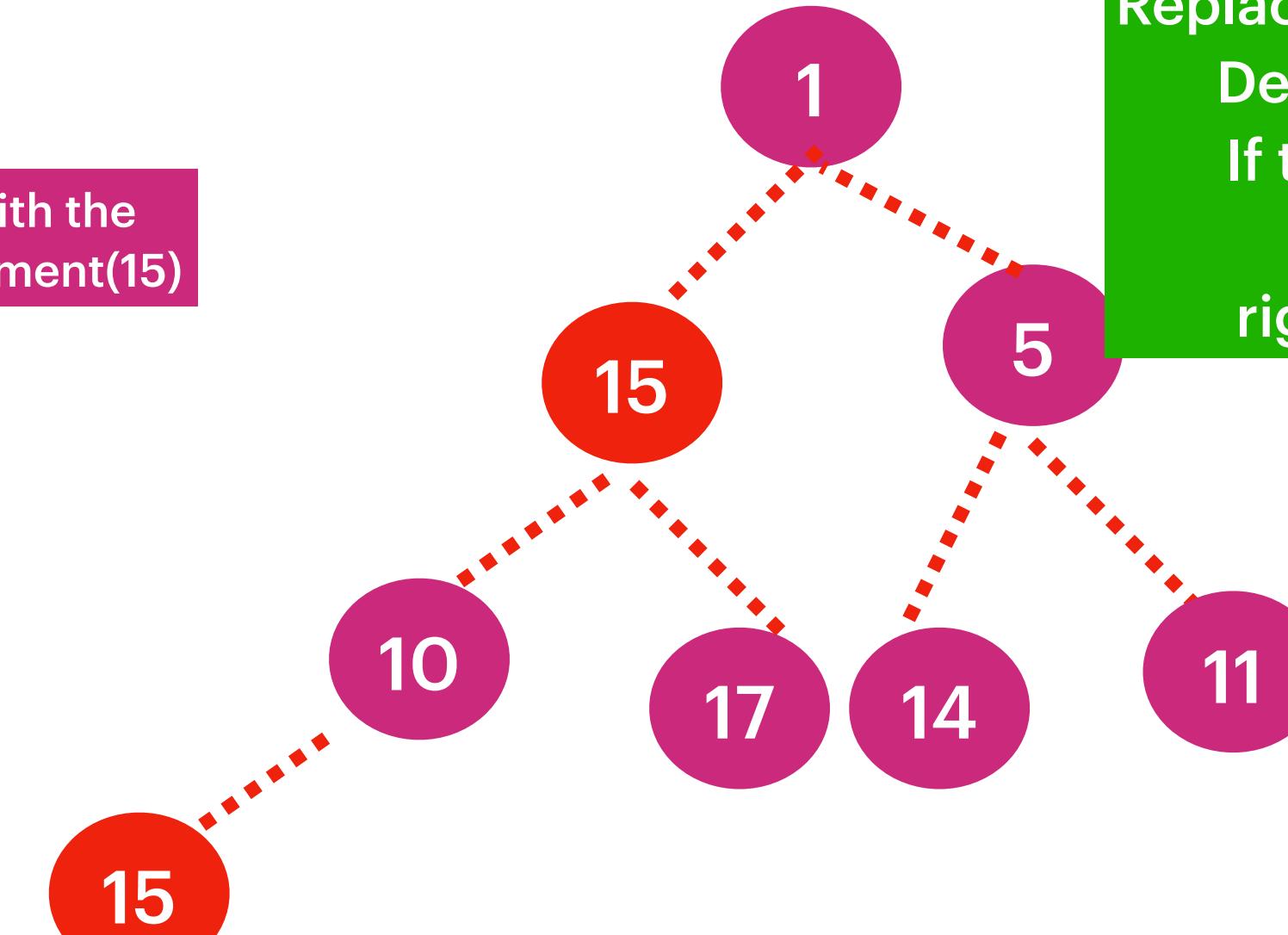
## Min Heap



## Delete Operation On MinHeap

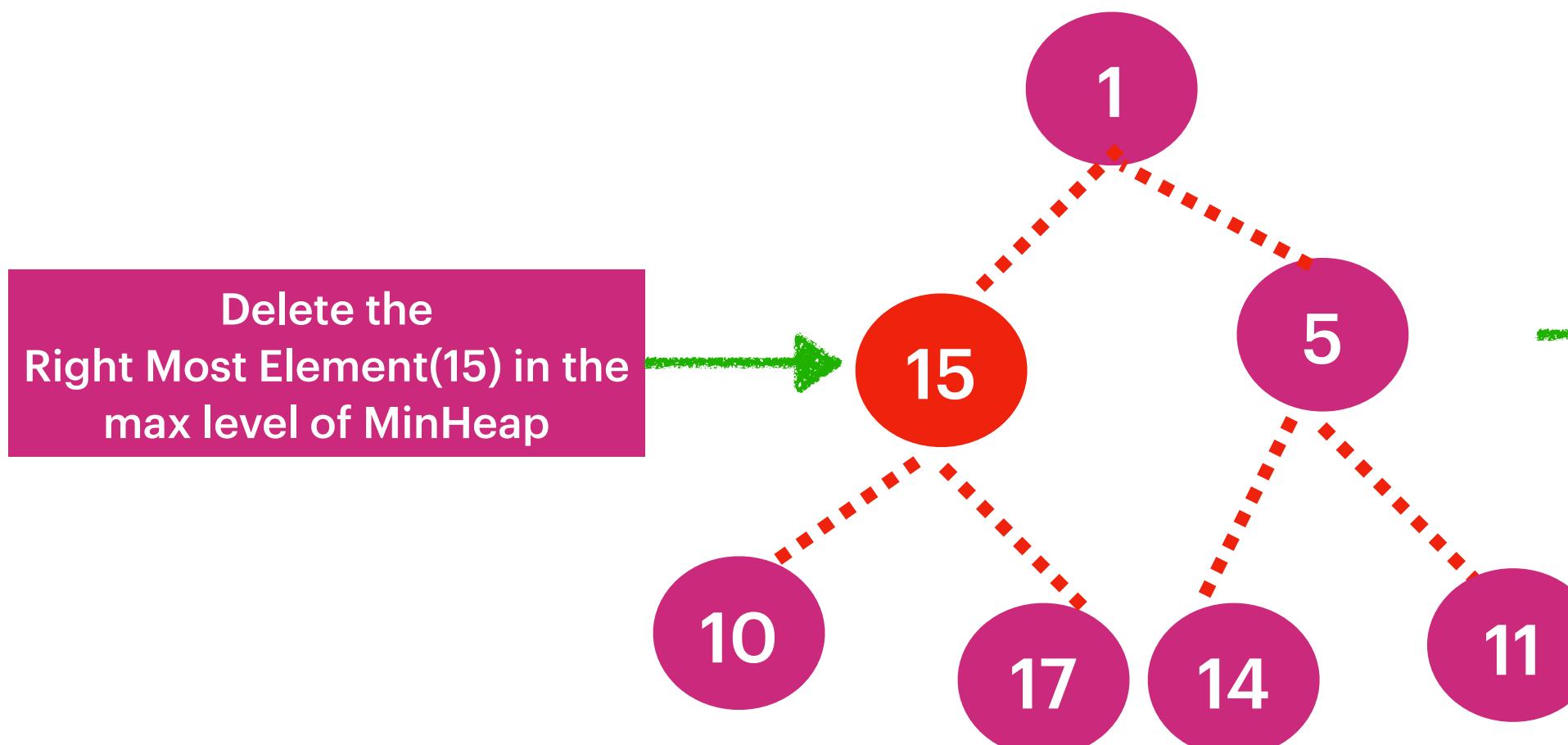
**delete(4)**

replace(4) with the Right Most Element(15)



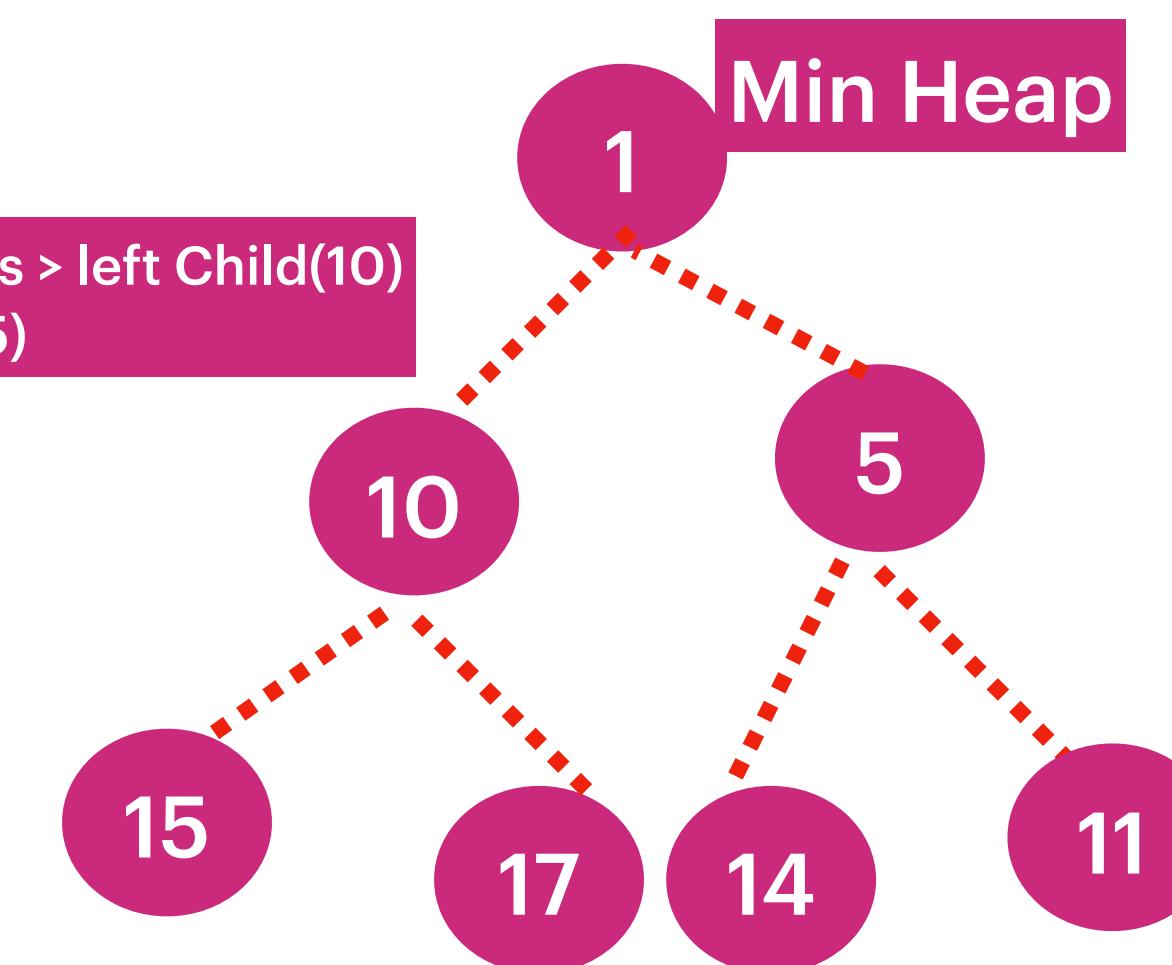
Algorithm :

Replace with the right most element ,  
Delete the right most element.  
If the current Element either >  
(leftNode || rightNode) swap accordingly

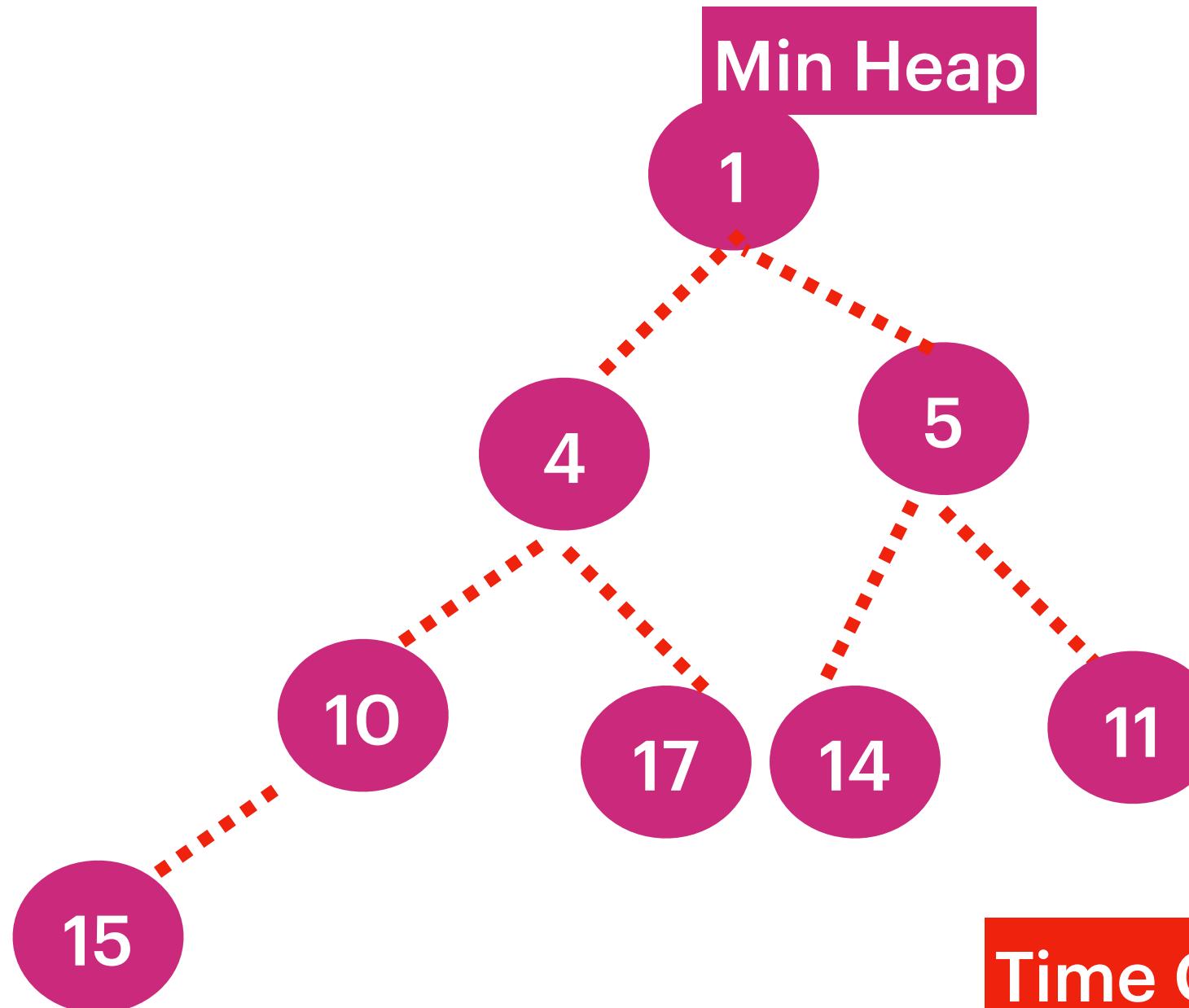


Delete the Right Most Element(15) in the max level of MinHeap

Current Parent(15) which is > left Child(10)  
Swap (10,15)



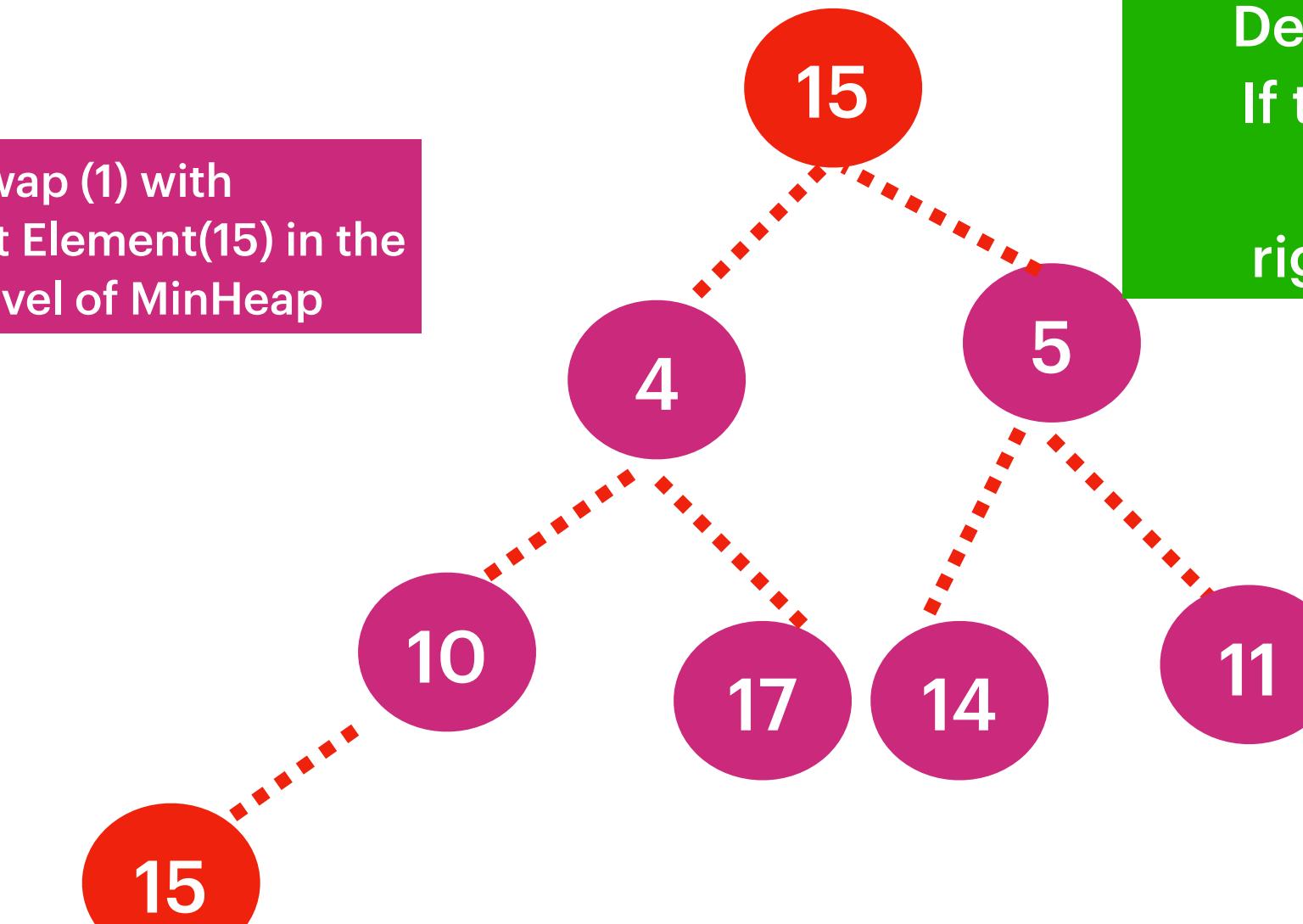
## Min Heap



**delete(1)**

## Delete Operation On MinHeap

Swap (1) with Right Most Element(15) in the max level of MinHeap



**Time Complexity : O(logn)**  
**Space Complexity : O(1)**

## Algorithm :

Replace with the right most element ,  
Delete the right most element.  
If the current Element either >  
(leftNode || rightNode) swap accordingly

