A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.

Construct a [**deep copy**](https://en.wikipedia.org/wiki/Object_copying#Deep_copy) of the list. The deep copy should consist of exactly n **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

For example, if there are two nodes X and Y in the original list, where X.random --> Y, then for the corresponding two nodes x and y in the copied list, x.random --> y.

Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random\_index] where:

* val: an integer representing Node.val
* random\_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node.

Your code will **only** be given the head of the original linked list.

**Example 1:**

Diagram

Description automatically generated

**Input:** head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

**Output:** [[7,null],[13,0],[11,4],[10,2],[1,0]]

**Example 2:**

A picture containing text, device

Description automatically generated

**Input:** head = [[1,1],[2,1]]

**Output:** [[1,1],[2,1]]

**Example 3:**

**Diagram

Description automatically generated**

**Input:** head = [[3,null],[3,0],[3,null]]

**Output:** [[3,null],[3,0],[3,null]]

**Constraints:**

* 0 <= n <= 1000
* -104 <= Node.val <= 104
* Node.random is null or is pointing to some node in the linked list.

Hint1:

* Just iterate the linked list and create copies of the nodes on the go. Since a node can be referenced from multiple nodes due to the random pointers, make sure you are not making multiple copies of the same node.
* You may want to use extra space to keep **old node ---> new node** mapping to prevent creating multiples copies of same node.
* We can avoid using extra space for old node ---> new node mapping, by tweaking the original linked list. Simply interweave the nodes of the old and copied list. For e.g.
* Old List: A --> B --> C --> D
* InterWeaved List: A --> A' --> B --> B' --> C --> C' --> D --> D'
* The interweaving is done using **next** pointers and we can make use of interweaved structure to get the correct reference nodes for **random** pointers.

/\*\*

\* // Definition for a Node.

\* function Node(val, next, random) {

\* this.val = val;

\* this.next = next;

\* this.random = random;

\* };

\*/

/\*\*

\* @param {Node} head

\* @return {Node}

\*/

var copyRandomList = function(head) {

};

Solution

Lets first look at how the linked list looks like

Diagram

Description automatically generated

In the above diagram, for a given node the next pointer points to the next node in the linked list. The next pointer is something standard for a linked list and this is what ***links*** the nodes together. What is interesting about the diagram and this problem is the random pointer which, as the name suggests can point to any node in the linked list or can be a null.

Approach 1: Recursive

**Intuition**

The basic idea behind the recursive solution is to consider the linked list like a graph. Every node of the Linked List has 2 pointers (edges in a graph). Since, random pointers add the randomness to the structure we might visit the same node again leading to cycles.

Diagram

Description automatically generated

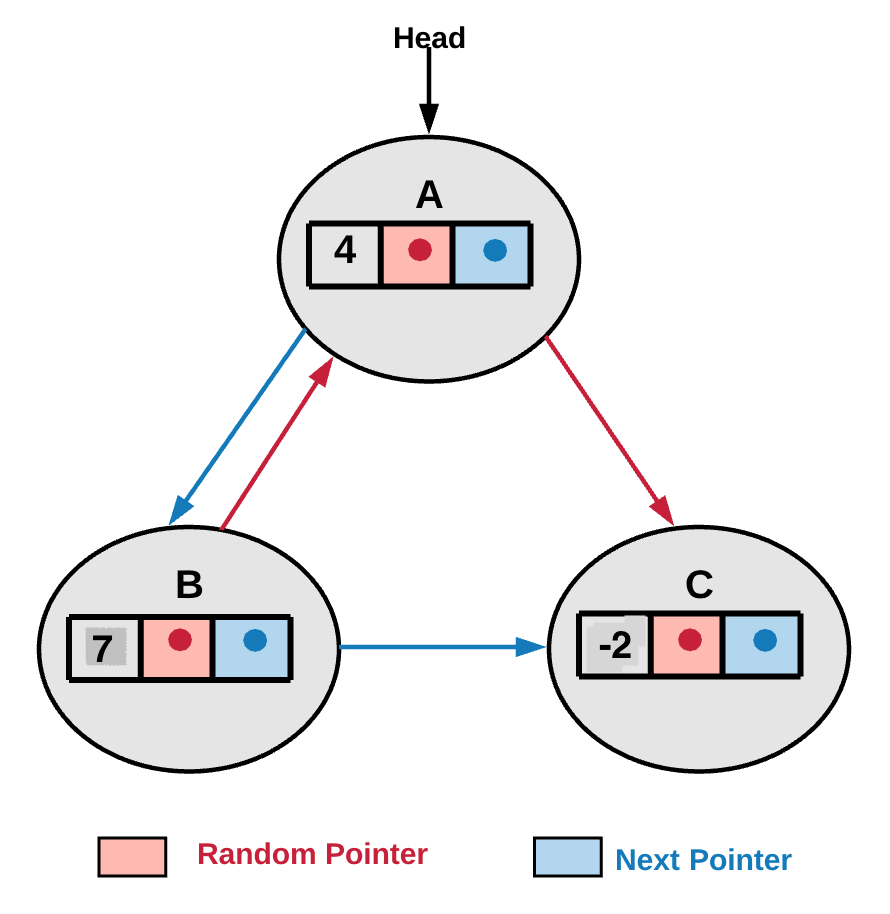
In the diagram above we can see the random pointer points back to the previously seen node hence leading to a cycle. We need to take care of these cycles in the implementation.

All we do in this approach is to just traverse the graph and clone it. Cloning essentially means creating a new node for every unseen node you encounter. The traversal part will happen recursively in a depth first manner. Note that we have to keep track of nodes already processed because, as pointed out earlier, we can have cycles because of the random pointers.

**Algorithm**

1. Start traversing the graph from head node.

Lets see the linked structure as a graph. Below is the graph representation of the above linked list example.



In the above example head is where we begin our graph traversal.

1. If we already have a cloned copy of the current node in the visited dictionary, we use the cloned node reference.
2. If we don't have a cloned copy in the visited dictionary, we create a new node and add it to the visited dictionary. visited\_dictionary[current\_node] = cloned\_node\_for\_current\_node.
3. We then make two recursive calls, one using the random pointer and the other using next pointer. The diagram from step 1, shows random and next pointers in red and blue color respectively. Essentially we are making recursive calls for the children of the current node. In this implementation, the children are the nodes pointed by the random and the next pointers.

cloned\_node\_for\_current\_node.next = copyRandomList(current\_node.next);

cloned\_node\_for\_current\_node.random = copyRandomList(current\_node.random);

/\*

// Definition for a Node.

class Node {

public int val;

public Node next;

public Node random;

public Node() {}

public Node(int \_val,Node \_next,Node \_random) {

val = \_val;

next = \_next;

random = \_random;

}

};

\*/

public class Solution {

// HashMap which holds old nodes as keys and new nodes as its values.

HashMap<Node, Node> visitedHash = new HashMap<Node, Node>();

public Node copyRandomList(Node head) {

if (head == null) {

return null;

}

// If we have already processed the current node, then we simply return the cloned version of

// it.

if (this.visitedHash.containsKey(head)) {

return this.visitedHash.get(head);

}

// Create a new node with the value same as old node. (i.e. copy the node)

Node node = new Node(head.val, null, null);

// Save this value in the hash map. This is needed since there might be

// loops during traversal due to randomness of random pointers and this would help us avoid

// them.

this.visitedHash.put(head, node);

// Recursively copy the remaining linked list starting once from the next pointer and then from

// the random pointer.

// Thus we have two independent recursive calls.

// Finally we update the next and random pointers for the new node created.

node.next = this.copyRandomList(head.next);

node.random = this.copyRandomList(head.random);

return node;

}

}

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*) where N is the number of nodes in the linked list.
* Space Complexity: O(N)*O*(*N*). If we look closely, we have the recursion stack and we also have the space complexity to keep track of nodes already cloned i.e. using the visited dictionary. But asymptotically, the complexity is O(N)*O*(*N*).

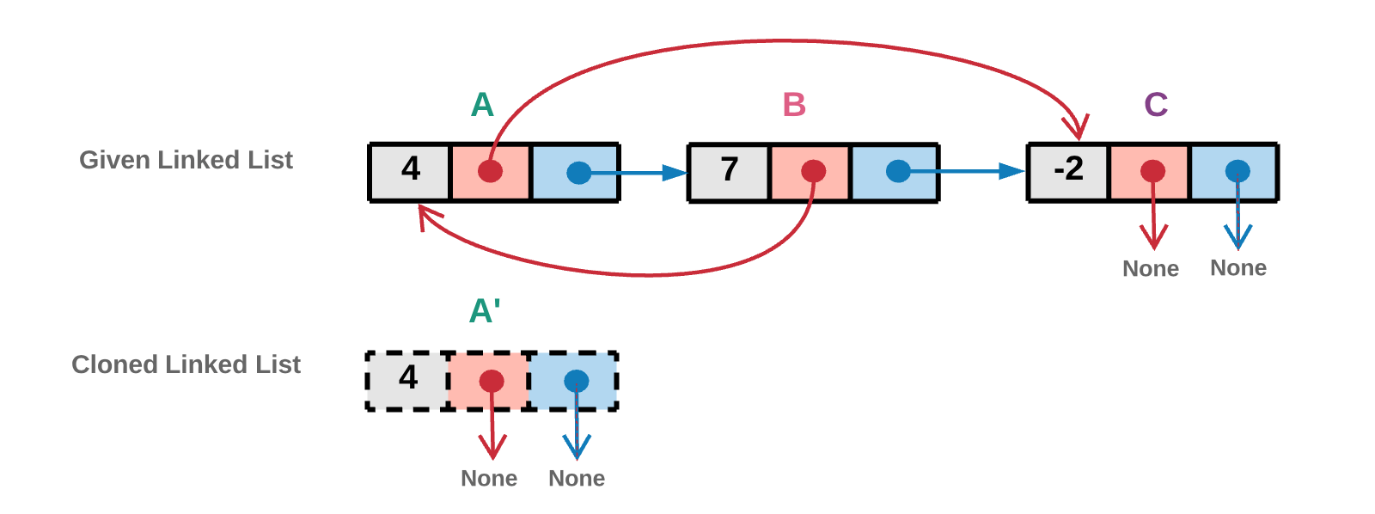
Approach 2: Iterative with O(N)*O*(*N*) Space

**Intuition**

The iterative solution to this problem does not model it as a graph, instead simply treats it as a LinkedList. When we are iterating over the list, we can create new nodes via the random pointer or the next pointer whichever points to a node that doesn't exist in our old --> new dictionary.

**Algorithm**

1. Traverse the linked list starting at head of the linked list.



In the above diagram we create a new cloned head node. The cloned node is shown using dashed lines. In the implementation we would even store the reference of this newly created node in a visited dictionary.

1. Random Pointer
   * If the random pointer of the current node i*i* points to the a node j*j* and a clone of j*j* already exists in the visited dictionary, we will simply use the cloned node reference from the visited dictionary.
   * If the random pointer of the current node i*i* points to the a node j*j* which has not been created yet, we create a new node corresponding to j*j* and add it to the visited dictionary.

Diagram

Description automatically generated

In the above diagram the random pointer of node A*A* points to a node C*C*. Node C*C* which was not visited yet as we can see from the previous diagram. Hence we create a new cloned C'*C*′ node corresponding to node C*C* and add it to visited dictionary.

1. Next Pointer
   * If the next pointer of the current node i*i* points to the a node j*j* and a clone of j*j* already exists in the visited dictionary, we will simply use the cloned node reference from the visited dictionary.
   * If the next pointer of the current node i*i* points to the a node j*j* which has not been created yet, we create a new node corresponding to j*j* and add it to the visited dictionary.

Diagram

Description automatically generated

In the above diagram the next pointer of node A*A* points to a node B*B*. Node B*B* which was not visited yet as we can see from the previous diagram. Hence we create a new cloned B'*B*′ node corresponding to node B*B* and add it to visited dictionary.

1. We repeat steps 2 and 3 until we reach the end of the linked list.

Diagram

Description automatically generated

In the above diagram, the random pointer of node B*B* points to an already visited node A*A*. Hence in step 2, we don't create a new copy for the clone. Instead we point random pointer of cloned node B'*B*′ to already existing cloned node A'*A*′.

Also, the next pointer of node B*B* points to an already visited node C*C*. Hence in step 3, we don't create a new copy for the clone. Instead we point next pointer of cloned node B'*B*′ to already existing cloned node C'*C*′.

/\*

// Definition for a Node.

class Node {

public int val;

public Node next;

public Node random;

public Node() {}

public Node(int \_val,Node \_next,Node \_random) {

val = \_val;

next = \_next;

random = \_random;

}

};

\*/

public class Solution {

// Visited dictionary to hold old node reference as "key" and new node reference as the "value"

HashMap<Node, Node> visited = new HashMap<Node, Node>();

public Node getClonedNode(Node node) {

// If the node exists then

if (node != null) {

// Check if the node is in the visited dictionary

if (this.visited.containsKey(node)) {

// If its in the visited dictionary then return the new node reference from the dictionary

return this.visited.get(node);

} else {

// Otherwise create a new node, add to the dictionary and return it

this.visited.put(node, new Node(node.val, null, null));

return this.visited.get(node);

}

}

return null;

}

public Node copyRandomList(Node head) {

if (head == null) {

return null;

}

Node oldNode = head;

// Creating the new head node.

Node newNode = new Node(oldNode.val);

this.visited.put(oldNode, newNode);

// Iterate on the linked list until all nodes are cloned.

while (oldNode != null) {

// Get the clones of the nodes referenced by random and next pointers.

newNode.random = this.getClonedNode(oldNode.random);

newNode.next = this.getClonedNode(oldNode.next);

// Move one step ahead in the linked list.

oldNode = oldNode.next;

newNode = newNode.next;

}

return this.visited.get(head);

}

}

/\*

// Definition for a Node.

class Node {

public int val;

public Node next;

public Node random;

public Node() {}

public Node(int \_val,Node \_next,Node \_random) {

val = \_val;

next = \_next;

random = \_random;

}

};

\*/

public class Solution {

// Visited dictionary to hold old node reference as "key" and new node reference as the "value"

HashMap<Node, Node> visited = new HashMap<Node, Node>();

public Node getClonedNode(Node node) {

// If the node exists then

if (node != null) {

// Check if the node is in the visited dictionary

if (this.visited.containsKey(node)) {

// If its in the visited dictionary then return the new node reference from the dictionary

return this.visited.get(node);

} else {

// Otherwise create a new node, add to the dictionary and return it

this.visited.put(node, new Node(node.val, null, null));

return this.visited.get(node);

}

}

return null;

}

public Node copyRandomList(Node head) {

if (head == null) {

return null;

}

Node oldNode = head;

// Creating the new head node.

Node newNode = new Node(oldNode.val);

this.visited.put(oldNode, newNode);

// Iterate on the linked list until all nodes are cloned.

while (oldNode != null) {

// Get the clones of the nodes referenced by random and next pointers.

newNode.random = this.getClonedNode(oldNode.random);

newNode.next = this.getClonedNode(oldNode.next);

// Move one step ahead in the linked list.

oldNode = oldNode.next;

newNode = newNode.next;

}

return this.visited.get(head);

}

}

**Complexity Analysis**

* Time Complexity : O(N)*O*(*N*) because we make one pass over the original linked list.
* Space Complexity : O(N)*O*(*N*) as we have a dictionary containing mapping from old list nodes to new list nodes. Since there are N*N* nodes, we have O(N)*O*(*N*) space complexity.

Approach 3: Iterative with O(1)*O*(1) Space

**Intuition**

Instead of a separate dictionary to keep the old node --> new node mapping, we can tweak the original linked list and keep every cloned node next to its original node. This interleaving of old and new nodes allows us to solve this problem without any extra space. Lets look at how the algorithm works.

**Algorithm**

1. Traverse the original list and clone the nodes as you go and place the cloned copy next to its original node. This new linked list is essentially a interweaving of original and cloned nodes.

Diagram

Description automatically generated

Chart

Description automatically generated

As you can see we just use the value of original node to create the cloned copy. The next pointer is used to create the weaving. Note that this operation ends up modifying the original linked list.

cloned\_node.next = original\_node.next

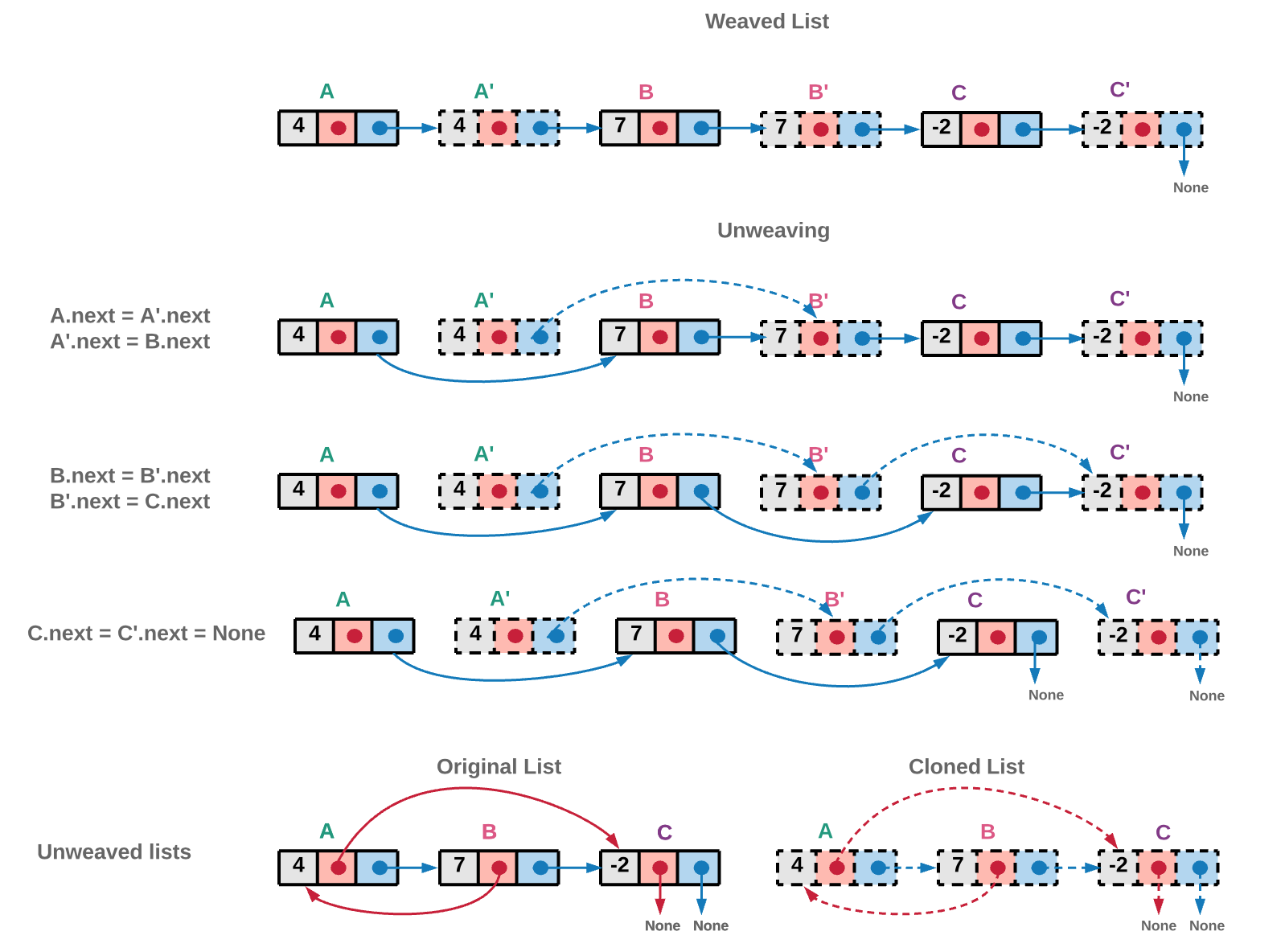
original\_node.next = cloned\_node

1. Iterate the list having both the new and old nodes intertwined with each other and use the original nodes' random pointers to assign references to random pointers for cloned nodes. For eg. If B has a random pointer to A, this means B' has a random pointer to A'.

Diagram

Description automatically generated

1. Now that the random pointers are assigned to the correct node, the next pointers need to be correctly assigned to unweave the current linked list and get back the original list and the cloned list.



/\*

// Definition for a Node.

class Node {

public int val;

public Node next;

public Node random;

public Node() {}

public Node(int \_val,Node \_next,Node \_random) {

val = \_val;

next = \_next;

random = \_random;

}

};

\*/

public class Solution {

public Node copyRandomList(Node head) {

if (head == null) {

return null;

}

// Creating a new weaved list of original and copied nodes.

Node ptr = head;

while (ptr != null) {

// Cloned node

Node newNode = new Node(ptr.val);

// Inserting the cloned node just next to the original node.

// If A->B->C is the original linked list,

// Linked list after weaving cloned nodes would be A->A'->B->B'->C->C'

newNode.next = ptr.next;

ptr.next = newNode;

ptr = newNode.next;

}

ptr = head;

// Now link the random pointers of the new nodes created.

// Iterate the newly created list and use the original nodes' random pointers,

// to assign references to random pointers for cloned nodes.

while (ptr != null) {

ptr.next.random = (ptr.random != null) ? ptr.random.next : null;

ptr = ptr.next.next;

}

// Unweave the linked list to get back the original linked list and the cloned list.

// i.e. A->A'->B->B'->C->C' would be broken to A->B->C and A'->B'->C'

Node ptr\_old\_list = head; // A->B->C

Node ptr\_new\_list = head.next; // A'->B'->C'

Node head\_old = head.next;

while (ptr\_old\_list != null) {

ptr\_old\_list.next = ptr\_old\_list.next.next;

ptr\_new\_list.next = (ptr\_new\_list.next != null) ? ptr\_new\_list.next.next : null;

ptr\_old\_list = ptr\_old\_list.next;

ptr\_new\_list = ptr\_new\_list.next;

}

return head\_old;

}

}

**Complexity Analysis**

* Time Complexity : O(N)*O*(*N*)
* Space Complexity : O(1)*O*(1)