Problem 426: Convert Binary Search Tree to Sorted Doubly Linked List

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Convert a

Binary Search Tree

to a sorted

Circular Doubly-Linked List

in place.

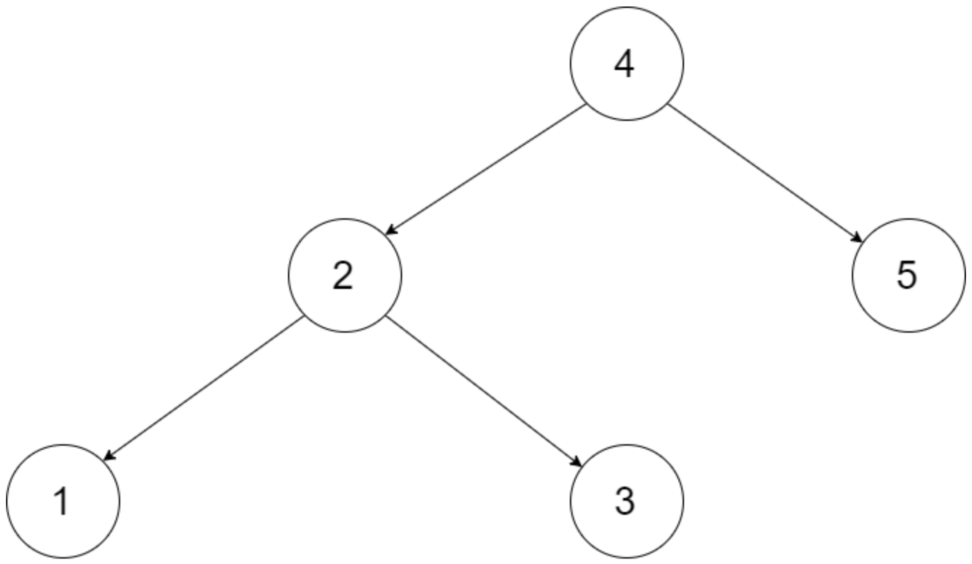
You can think of the left and right pointers as synonymous to the predecessor and successor pointers in a doubly-linked list. For a circular doubly linked list, the predecessor of the first element is the last element, and the successor of the last element is the first element.

We want to do the transformation

in place

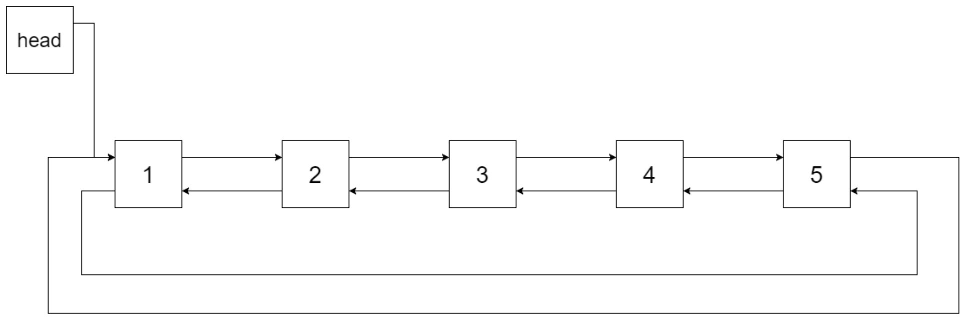
. After the transformation, the left pointer of the tree node should point to its predecessor, and the right pointer should point to its successor. You should return the pointer to the smallest element of the linked list.

Example 1:



Input:

root = [4,2,5,1,3]

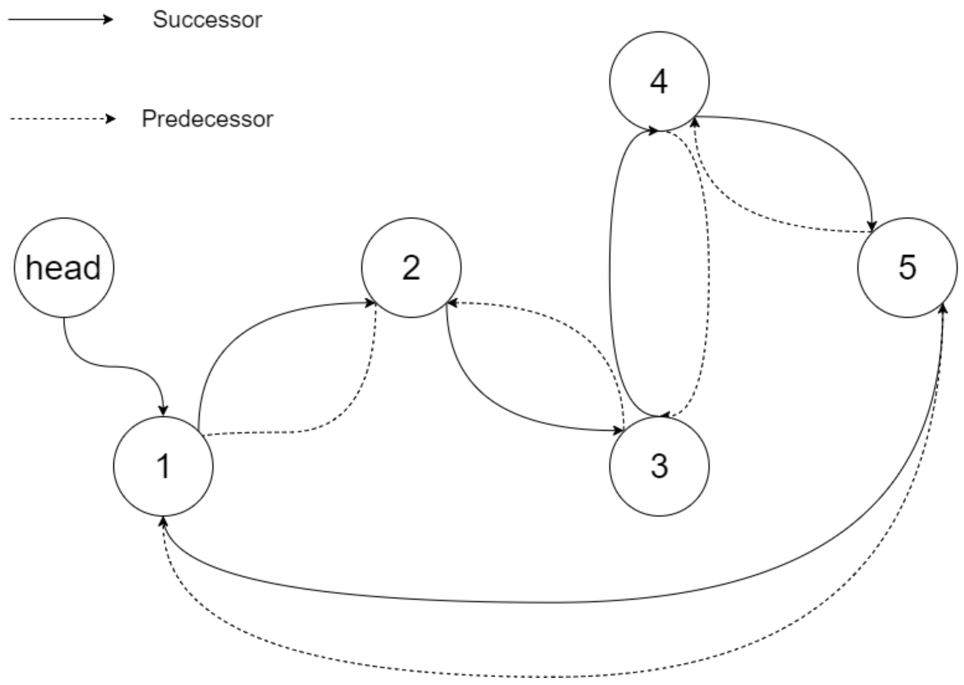


Output:

[1,2,3,4,5]

Explanation:

The figure below shows the transformed BST. The solid line indicates the successor relationship, while the dashed line means the predecessor relationship.



Example 2:

Input:

root = [2,1,3]

Output:

[1,2,3]

Constraints:

The number of nodes in the tree is in the range

[0, 2000]

.

-1000 <= Node.val <= 1000

All the values of the tree are

unique

.

Code Snippets

C++:

```
/*
// Definition for a Node.
class Node {
public:
int val;
Node* left;
Node* right;

Node() {}

Node(int _val) {
val = _val;
left = NULL;
right = NULL;
}

Node(int _val, Node* _left, Node* _right) {
val = _val;
left = _left;
right = _right;
}
};
*/

class Solution {
public:
Node* treeToDoublyList(Node* root) {

}
};
```

Java:

```
/*
// Definition for a Node.
```

```
class Node {
public int val;
public Node left;
public Node right;

public Node() {}

public Node(int _val) {
val = _val;
}

public Node(int _val,Node _left,Node _right) {
val = _val;
left = _left;
right = _right;
}
};
*/

class Solution {
public Node treeToDoublyList(Node root) {

}
}
```

Python3:

```
"""
# Definition for a Node.
class Node:
def __init__(self, val, left=None, right=None):
self.val = val
self.left = left
self.right = right
"""

class Solution:
def treeToDoublyList(self, root: 'Optional[Node]') -> 'Optional[Node]':
```

Python:

```
"""
# Definition for a Node.
class Node(object):
def __init__(self, val, left=None, right=None):
self.val = val
self.left = left
self.right = right
"""

class Solution(object):
def treeToDoublyList(self, root):
"""
:type root: Node
:rtype: Node
"""
```

JavaScript:

```
/**
* // Definition for a _Node.
* function _Node(val, left, right) {
* this.val = val;
* this.left = left;
* this.right = right;
* };
*/

/**
* @param {_Node} root
* @return {_Node}
*/
var treeToDoublyList = function(root) {

};
```

TypeScript:

```
/**
* Definition for _Node.
* class _Node {
* val: number
* left: _Node | null
* right: _Node | null
*
* constructor(val?: number, left?: _Node | null, right?: _Node | null) {
* this.val = (val===undefined ? 0 : val)
* this.left = (left===undefined ? null : left)
* this.right = (right===undefined ? null : right)
* }
* }
*/


function treeToDoublyList(root: _Node | null): _Node | null {

};
```

C#:

```
/*
// Definition for a Node.
public class Node {
public int val;
public Node left;
public Node right;

public Node() {}

public Node(int _val) {
val = _val;
left = null;
right = null;
}

public Node(int _val,Node _left,Node _right) {
val = _val;
left = _left;
right = _right;
}
}
*/

public class Solution {
public Node TreeToDoublyList(Node root) {

}
}
```

C:

```
/*
// Definition for a Node.
struct Node {
int val;
struct Node* left;
struct Node* right;
};
*/

struct Node* treeToDoublyList(struct Node *root) {

}
```

Go:

```
/**
* Definition for a Node.
* type Node struct {
* Val int
* Left *Node
* Right *Node
* }
*/

func treeToDoublyList(root *Node) *Node {

}
```

Kotlin:

```
/**
* Definition for a Node.
* class Node(var `val`: Int) {
* var left: Node? = null
* var right: Node? = null
* }
*/
```

```
class Solution {
fun treeToDoublyList(root:Node?): Node? {

}
}
```

Swift:

```
/**
* Definition for a Node.
* public class Node {
* public var val: Int
* public var left: Node?
* public var right: Node?
* public init(_ val: Int) {
* self.val = val
* self.left = nil
* self.right = nil
* }
* }
*/

class Solution {
func treeToDoublyList(_ root: Node?) -> Node? {

}
}
```

Ruby:

```
# Definition for a Node.
# class Node
# attr_accessor :val, :left, :right
# def initialize(val=0)
# @val = val
# @left, @right = nil, nil
# end
# end

# @param {Node} root
# @return {Node}
```

```
def treeToDoublyList(root)

end
```

PHP:

```
/**
* Definition for a Node.
* class Node {
* public $val = null;
* public $left = null;
* public $right = null;
* function __construct($val = 0) {
* $this->val = $val;
* $this->left = null;
* $this->right = null;
* }
* }
*/

class Solution {
/**
* @param Node $root
* @return Node
*/
function treeToDoublyList($root) {

}
}
```

Scala:

```
/**
* Definition for a Node.
* class Node(var _value: Int) {
* var value: Int = _value
* var left: Node = null
* var right: Node = null
* }
*/

object Solution {
```

```
def treeToDoublyList(root: Node): Node = {

}
}
```

Solutions

C++ Solution:

```
/*
* Problem: Convert Binary Search Tree to Sorted Doubly Linked List
* Difficulty: Medium
* Tags: tree, sort, search, linked_list, stack
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a Node.
class Node {
public:
int val;
Node* left;
Node* right;

Node() {}

Node(int _val) {
val = _val;
left = NULL;
right = NULL;
}

Node(int _val, Node* _left, Node* _right) {
val = _val;
left = _left;
right = _right;
}
};
*/

class Solution {
public:
Node* treeToDoublyList(Node* root) {

}
};
```

Java Solution:

```
/**
* Problem: Convert Binary Search Tree to Sorted Doubly Linked List
* Difficulty: Medium
* Tags: tree, sort, search, linked_list, stack
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a Node.
class Node {
public int val;
public Node left;
public Node right;

public Node() {
// TODO: Implement optimized solution
return 0;
}

public Node(int _val) {
val = _val;
}

public Node(int _val,Node _left,Node _right) {
val = _val;
left = _left;
right = _right;
}
};
*/

class Solution {
public Node treeToDoublyList(Node root) {

}
}
```

Python3 Solution:

```
"""
Problem: Convert Binary Search Tree to Sorted Doubly Linked List
Difficulty: Medium
Tags: tree, sort, search, linked_list, stack

Approach: DFS or BFS traversal
Time Complexity: O(n) where n is number of nodes
Space Complexity: O(h) for recursion stack where h is height
"""

"""
# Definition for a Node.
class Node:
def __init__(self, val, left=None, right=None):
self.val = val
self.left = left
self.right = right
"""

class Solution:
def treeToDoublyList(self, root: 'Optional[Node]') -> 'Optional[Node]':
# TODO: Implement optimized solution
pass
```

Python Solution:

```
"""
# Definition for a Node.
```

```
class Node(object):
def __init__(self, val, left=None, right=None):
self.val = val
self.left = left
self.right = right
"""

class Solution(object):
def treeToDoublyList(self, root):
"""
:type root: Node
:rtype: Node
"""
```

JavaScript Solution:

```
/**
* Problem: Convert Binary Search Tree to Sorted Doubly Linked List
* Difficulty: Medium
* Tags: tree, sort, search, linked_list, stack
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* // Definition for a _Node.
* function _Node(val, left, right) {
* this.val = val;
* this.left = left;
* this.right = right;
* };
*/

/**
* @param {_Node} root
* @return {_Node}
*/
var treeToDoublyList = function(root) {
};
```

TypeScript Solution:

```
/**
* Problem: Convert Binary Search Tree to Sorted Doubly Linked List
* Difficulty: Medium
* Tags: tree, sort, search, linked_list, stack
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Definition for _Node.
* class _Node {
* val: number
* left: _Node | null
* right: _Node | null
*
* constructor(val?: number, left?: _Node | null, right?: _Node | null) {
* this.val = (val===undefined ? 0 : val)
* this.left = (left===undefined ? null : left)
* this.right = (right===undefined ? null : right)
* }
* }
*/


function treeToDoublyList(root: _Node | null): _Node | null {

};
```

C# Solution:

```
/*
* Problem: Convert Binary Search Tree to Sorted Doubly Linked List
* Difficulty: Medium
* Tags: tree, sort, search, linked_list, stack
*
```

```
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a Node.
public class Node {
public int val;
public Node left;
public Node right;

public Node() {}

public Node(int _val) {
val = _val;
left = null;
right = null;
}

public Node(int _val,Node _left,Node _right) {
val = _val;
left = _left;
right = _right;
}
}
*/

public class Solution {
public Node TreeToDoublyList(Node root) {

}
}
```

C Solution:

```
/*
* Problem: Convert Binary Search Tree to Sorted Doubly Linked List
* Difficulty: Medium
* Tags: tree, sort, search, linked_list, stack
*
```

```
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a Node.
struct Node {
int val;
struct Node* left;
struct Node* right;
};
*/

struct Node* treeToDoublyList(struct Node *root) {

}
```

Go Solution:

```
// Problem: Convert Binary Search Tree to Sorted Doubly Linked List
// Difficulty: Medium
// Tags: tree, sort, search, linked_list, stack
//
// Approach: DFS or BFS traversal
// Time Complexity: O(n) where n is number of nodes
// Space Complexity: O(h) for recursion stack where h is height

/**
* Definition for a Node.
* type Node struct {
* Val int
* Left *Node
* Right *Node
* }
*/

func treeToDoublyList(root *Node) *Node {

}
```

Kotlin Solution:

```
/**
* Definition for a Node.
* class Node(var `val`: Int) {
* var left: Node? = null
* var right: Node? = null
* }
*/

class Solution {
fun treeToDoublyList(root:Node?): Node? {

}
}
```

Swift Solution:

```
/**
* Definition for a Node.
* public class Node {
* public var val: Int
* public var left: Node?
* public var right: Node?
* public init(_ val: Int) {
* self.val = val
* self.left = nil
* self.right = nil
* }
* }
*/

class Solution {
func treeToDoublyList(_ root: Node?) -> Node? {

}
}
```

Ruby Solution:

```
# Definition for a Node.
# class Node
# attr_accessor :val, :left, :right
# def initialize(val=0)
# @val = val
# @left, @right = nil, nil
# end
# end

# @param {Node} root
# @return {Node}
def treeToDoublyList(root)

end
```

PHP Solution:

```
/**
* Definition for a Node.
* class Node {
* public $val = null;
* public $left = null;
* public $right = null;
* function __construct($val = 0) {
* $this->val = $val;
* $this->left = null;
* $this->right = null;
* }
* }
*/

class Solution {
/**
* @param Node $root
* @return Node
*/
function treeToDoublyList($root) {

}
}
```

Scala Solution:

```
/**
* Definition for a Node.
* class Node(var _value: Int) {
* var value: Int = _value
* var left: Node = null
* var right: Node = null
* }
*/

object Solution {
def treeToDoublyList(root: Node): Node = {

}
}
```