Problem 430: Flatten a Multilevel Doubly Linked List

Problem Information

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

Problem Description

You are given a doubly linked list, which contains nodes that have a next pointer, a previous pointer, and an additional

child pointer

. This child pointer may or may not point to a separate doubly linked list, also containing these special nodes. These child lists may have one or more children of their own, and so on, to produce a

multilevel data structure

as shown in the example below.

Given the

head

of the first level of the list,

flatten

the list so that all the nodes appear in a single-level, doubly linked list. Let

curr

be a node with a child list. The nodes in the child list should appear

after

curr

and

before

curr.next

in the flattened list.

Return

the

head

of the flattened list. The nodes in the list must have

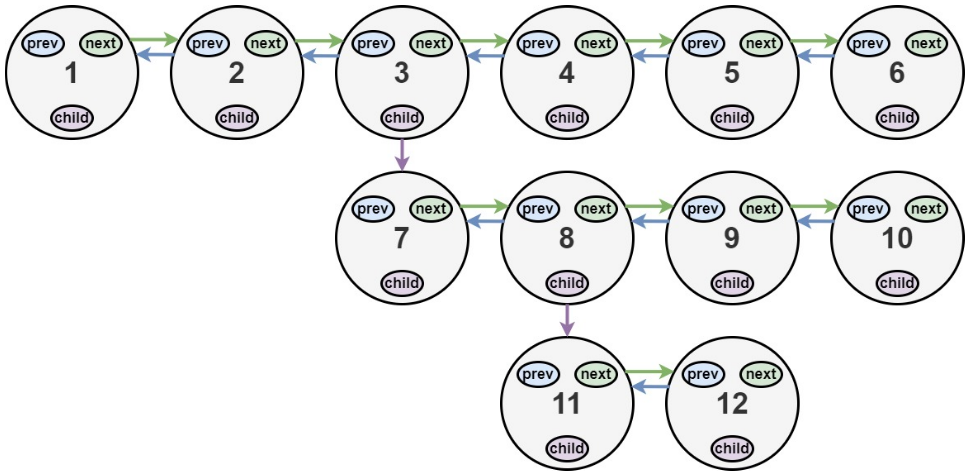
all

of their child pointers set to

null

.

Example 1:



Input:

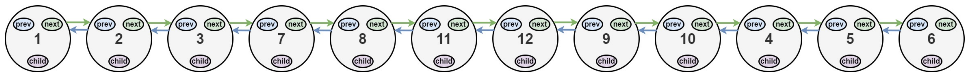
head = [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]

Output:

[1,2,3,7,8,11,12,9,10,4,5,6]

Explanation:

The multilevel linked list in the input is shown. After flattening the multilevel linked list it becomes:



Example 2:

Input:

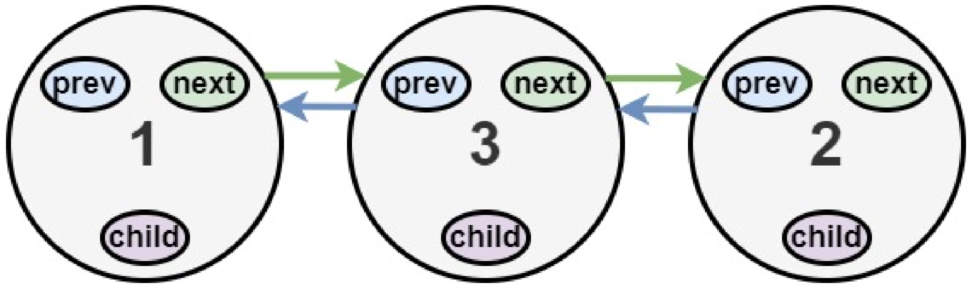
head = [1,2,null,3]

Output:

[1,3,2]

Explanation:

The multilevel linked list in the input is shown. After flattening the multilevel linked list it becomes:



Example 3:

Input:

head = []

Output:

[]

Explanation:

There could be empty list in the input.

Constraints:

The number of Nodes will not exceed

1000

.

1 <= Node.val <= 10

5

How the multilevel linked list is represented in test cases:

We use the multilevel linked list from

Example 1

above:

1---2---3---4---5---6--NULL | 7---8---9---10--NULL | 11--12--NULL

The serialization of each level is as follows:

[1,2,3,4,5,6,null] [7,8,9,10,null] [11,12,null]

To serialize all levels together, we will add nulls in each level to signify no node connects to the upper node of the previous level. The serialization becomes:

[1, 2, 3, 4, 5, 6, null] | [null, null, 7, 8, 9, 10, null] | [null, 11, 12, null]

Merging the serialization of each level and removing trailing nulls we obtain:

[1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]

Code Snippets

C++:

```
/*
// Definition for a Node.
class Node {
public:
int val;
Node* prev;
Node* next;
Node* child;
};
*/

class Solution {
public:
Node* flatten(Node* head) {

}
};
```

Java:

```
/*
// Definition for a Node.
class Node {
public int val;
public Node prev;
public Node next;
public Node child;
};
*/

class Solution {
public Node flatten(Node head) {

}
}
```

Python3:

```
"""
# Definition for a Node.
class Node:
def __init__(self, val, prev, next, child):
self.val = val
self.prev = prev
self.next = next
self.child = child
"""

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

Python:

```
"""
# Definition for a Node.
class Node(object):
def __init__(self, val, prev, next, child):
self.val = val
self.prev = prev
self.next = next
self.child = child
"""

class Solution(object):
def flatten(self, head):
"""
:type head: Node
:rtype: Node
"""
```

JavaScript:

```
/**
* // Definition for a _Node.
* function _Node(val,prev,next,child) {
* this.val = val;
* this.prev = prev;
* this.next = next;
* this.child = child;
* };
*/

/**
* @param {_Node} head
* @return {_Node}
*/
var flatten = function(head) {

};
```

TypeScript:

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


function flatten(head: _Node | null): _Node | null {

};
```

C#:

```
/*
// Definition for a Node.
public class Node {
public int val;
public Node prev;
public Node next;
public Node child;
}
*/

public class Solution {
public Node Flatten(Node head) {

}
}
```

Go:

```
/**
* Definition for a Node.
* type Node struct {
* Val int
* Prev *Node
* Next *Node
* Child *Node
* }
```

```
*/

func flatten(root *Node) *Node {

}
```

Kotlin:

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

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

}
}
```

Swift:

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

class Solution {
```

```
func flatten(_ head: Node?) -> Node? {

}
}
```

Ruby:

```
# Definition for a Node.
# class Node
# attr_accessor :val, :prev, :next, :child
# def initialize(val=nil, prev=nil, next_=nil, child=nil)
# @val = val
# @prev = prev
# @next = next_
# @child = child
# end
# end

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

end
```

PHP:

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

```
class Solution {
/**
* @param Node $head
* @return Node
*/
function flatten($head) {

}
}
```

Scala:

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

object Solution {
def flatten(head: Node): Node = {

}
}
```

Solutions

C++ Solution:

```
/*
* Problem: Flatten a Multilevel Doubly Linked List
* Difficulty: Medium
* Tags: search, linked_list
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
```

```
*/

/*
// Definition for a Node.
class Node {
public:
int val;
Node* prev;
Node* next;
Node* child;
};
*/

class Solution {
public:
Node* flatten(Node* head) {

}
};
```

Java Solution:

```
/**
* Problem: Flatten a Multilevel Doubly Linked List
* Difficulty: Medium
* Tags: search, linked_list
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

/*
// Definition for a Node.
class Node {
public int val;
public Node prev;
public Node next;
public Node child;
};
*/
```

```
class Solution {
public Node flatten(Node head) {

}
}
```

Python3 Solution:

```
"""
Problem: Flatten a Multilevel Doubly Linked List
Difficulty: Medium
Tags: search, linked_list

Approach: Optimized algorithm based on problem constraints
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(1) to O(n) depending on approach
"""

"""
# Definition for a Node.
class Node:
def __init__(self, val, prev, next, child):
self.val = val
self.prev = prev
self.next = next
self.child = child
"""

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

Python Solution:

```
"""
# Definition for a Node.
class Node(object):
def __init__(self, val, prev, next, child):
self.val = val
self.prev = prev
self.next = next
self.child = child
"""

class Solution(object):
def flatten(self, head):
"""
:type head: Node
:rtype: Node
"""
```

JavaScript Solution:

```
/**
* Problem: Flatten a Multilevel Doubly Linked List
* Difficulty: Medium
* Tags: search, linked_list
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* // Definition for a _Node.
* function _Node(val,prev,next,child) {
* this.val = val;
* this.prev = prev;
* this.next = next;
* this.child = child;
* };
*/

/**
* @param {_Node} head
* @return {_Node}
*/
var flatten = function(head) {

};
```

TypeScript Solution:

```
/**
* Problem: Flatten a Multilevel Doubly Linked List
* Difficulty: Medium
* Tags: search, linked_list
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* Definition for _Node.
* class _Node {
* val: number
* prev: _Node | null
* next: _Node | null
* child: _Node | null
*
* constructor(val?: number, prev? : _Node, next? : _Node, child? : _Node) {
* this.val = (val===undefined ? 0 : val);
* this.prev = (prev===undefined ? null : prev);
* this.next = (next===undefined ? null : next);
* this.child = (child===undefined ? null : child);
* }
* }
*/



function flatten(head: _Node | null): _Node | null {

};
```

C# Solution:

```
/*
* Problem: Flatten a Multilevel Doubly Linked List
* Difficulty: Medium
* Tags: search, linked_list
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

/*
// Definition for a Node.
public class Node {
public int val;
public Node prev;
public Node next;
public Node child;
}
*/

public class Solution {
public Node Flatten(Node head) {

}
}
```

Go Solution:

```
// Problem: Flatten a Multilevel Doubly Linked List
// Difficulty: Medium
// Tags: search, linked_list
//
// Approach: Optimized algorithm based on problem constraints
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(1) to O(n) depending on approach

/**
* Definition for a Node.
* type Node struct {
* Val int
* Prev *Node
* Next *Node
* Child *Node
* }
*/

func flatten(root *Node) *Node {
}
```

Kotlin Solution:

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

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

}
}
```

Swift Solution:

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

class Solution {
func flatten(_ head: Node?) -> Node? {
}
}
```

Ruby Solution:

```
# Definition for a Node.
# class Node
# attr_accessor :val, :prev, :next, :child
# def initialize(val=nil, prev=nil, next_=nil, child=nil)
# @val = val
# @prev = prev
# @next = next_
# @child = child
# end
# end

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

end
```

PHP Solution:

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

class Solution {
```

```
/**
* @param Node $head
* @return Node
*/
function flatten($head) {

}
}
```

Scala Solution:

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

object Solution {
def flatten(head: Node): Node = {

}
}
```