Problem 1993: Operations on Tree

Problem Information

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

Problem Description

You are given a tree with

n

nodes numbered from

0

to

n - 1

in the form of a parent array

parent

where

parent[i]

is the parent of the

i

th

node. The root of the tree is node

0

, so

parent[0] = -1

since it has no parent. You want to design a data structure that allows users to lock, unlock, and upgrade nodes in the tree.

The data structure should support the following functions:

Lock:

Locks

the given node for the given user and prevents other users from locking the same node. You may only lock a node using this function if the node is unlocked.

Unlock: Unlocks

the given node for the given user. You may only unlock a node using this function if it is currently locked by the same user.

Upgrade

: Locks

the given node for the given user and

unlocks

all of its descendants

regardless

of who locked it. You may only upgrade a node if

all

3 conditions are true:

The node is unlocked,

It has at least one locked descendant (by

any

user), and

It does not have any locked ancestors.

Implement the

LockingTree

class:

LockingTree(int[] parent)

initializes the data structure with the parent array.

lock(int num, int user)

returns

true

if it is possible for the user with id

user

to lock the node

num

, or

false

otherwise. If it is possible, the node

num

will become

locked

by the user with id

user

.

unlock(int num, int user)

returns

true

if it is possible for the user with id

user

to unlock the node

num

, or

false

otherwise. If it is possible, the node

num

will become

unlocked

.

upgrade(int num, int user)

returns

true

if it is possible for the user with id

user

to upgrade the node

num

, or

false

otherwise. If it is possible, the node

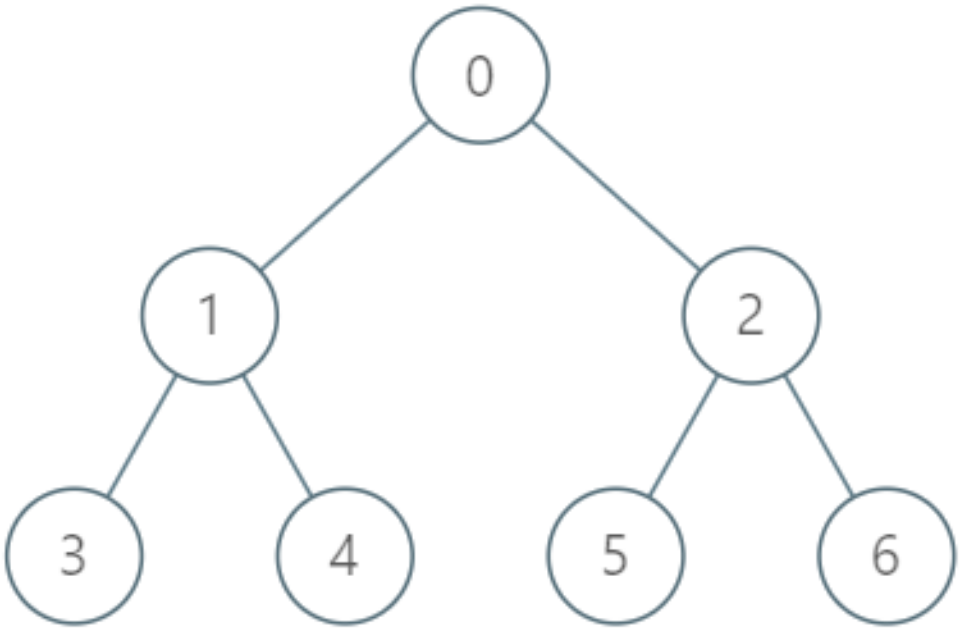
num

will be

upgraded

.

Example 1:



Input

["LockingTree", "lock", "unlock", "unlock", "lock", "upgrade", "lock"] [[[-1, 0, 0, 1, 1, 2, 2]], [2, 2], [2, 3], [2, 2], [4, 5], [0, 1], [0, 1]]

Output

[null, true, false, true, true, true, false]

Explanation

LockingTree lockingTree = new LockingTree([-1, 0, 0, 1, 1, 2, 2]); lockingTree.lock(2, 2); // return true because node 2 is unlocked. // Node 2 will now be locked by user 2. lockingTree.unlock(2, 3); // return false because user 3 cannot unlock a node locked by user 2. lockingTree.unlock(2, 2); // return true because node 2 was previously locked by user 2. // Node 2 will now be unlocked. lockingTree.lock(4, 5); // return true because node 4 is unlocked. // Node 4 will now be locked by user 5. lockingTree.upgrade(0, 1); // return true because node 0 is unlocked and has at least one locked descendant (node 4). // Node 0 will now be locked by user 1 and node 4 will now be unlocked. lockingTree.lock(0, 1); // return false because node 0 is already locked.

Constraints:

n == parent.length

2 <= n <= 2000

0 <= parent[i] <= n - 1

for

i != 0

parent[0] == -1

0 <= num <= n - 1

1 <= user <= 10

4

parent

represents a valid tree.

At most

2000

calls

in total

will be made to

lock

,

unlock

, and

upgrade

.

Code Snippets

C++:

```
class LockingTree {
public:
LockingTree(vector<int>& parent) {

}

bool lock(int num, int user) {

}

bool unlock(int num, int user) {

}

bool upgrade(int num, int user) {

}
};

/**
* Your LockingTree object will be instantiated and called as such:
* LockingTree* obj = new LockingTree(parent);
* bool param_1 = obj->lock(num,user);
* bool param_2 = obj->unlock(num,user);
* bool param_3 = obj->upgrade(num,user);
*/
```

Java:

```
class LockingTree {

public LockingTree(int[] parent) {

}
```

```
public boolean lock(int num, int user) {

}

public boolean unlock(int num, int user) {

}

public boolean upgrade(int num, int user) {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* LockingTree obj = new LockingTree(parent);
* boolean param_1 = obj.lock(num,user);
* boolean param_2 = obj.unlock(num,user);
* boolean param_3 = obj.upgrade(num,user);
*/
```

Python3:

```
class LockingTree:

def __init__(self, parent: List[int]):


def lock(self, num: int, user: int) -> bool:


def unlock(self, num: int, user: int) -> bool:


def upgrade(self, num: int, user: int) -> bool:



# Your LockingTree object will be instantiated and called as such:
# obj = LockingTree(parent)
# param_1 = obj.lock(num,user)
# param_2 = obj.unlock(num,user)
```

```
# param_3 = obj.upgrade(num,user)
```

Python:

```
class LockingTree(object):

def __init__(self, parent):
"""
:type parent: List[int]
"""


def lock(self, num, user):
"""
:type num: int
:type user: int
:rtype: bool
"""


def unlock(self, num, user):
"""
:type num: int
:type user: int
:rtype: bool
"""


def upgrade(self, num, user):
"""
:type num: int
:type user: int
:rtype: bool
"""



# Your LockingTree object will be instantiated and called as such:
# obj = LockingTree(parent)
# param_1 = obj.lock(num,user)
# param_2 = obj.unlock(num,user)
# param_3 = obj.upgrade(num,user)
```

JavaScript:

```
/**
* @param {number[]} parent
*/
var LockingTree = function(parent) {

};

/**
* @param {number} num
* @param {number} user
* @return {boolean}
*/
LockingTree.prototype.lock = function(num, user) {

};

/**
* @param {number} num
* @param {number} user
* @return {boolean}
*/
LockingTree.prototype.unlock = function(num, user) {

};

/**
* @param {number} num
* @param {number} user
* @return {boolean}
*/
LockingTree.prototype.upgrade = function(num, user) {

};

/**
* Your LockingTree object will be instantiated and called as such:
* var obj = new LockingTree(parent)
* var param_1 = obj.lock(num,user)
* var param_2 = obj.unlock(num,user)
* var param_3 = obj.upgrade(num,user)
```

```
*/
```

TypeScript:

```
class LockingTree {
constructor(parent: number[]) {

}

lock(num: number, user: number): boolean {

}

unlock(num: number, user: number): boolean {

}

upgrade(num: number, user: number): boolean {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* var obj = new LockingTree(parent)
* var param_1 = obj.lock(num,user)
* var param_2 = obj.unlock(num,user)
* var param_3 = obj.upgrade(num,user)
*/
```

C#:

```
public class LockingTree {

public LockingTree(int[] parent) {

}

public bool Lock(int num, int user) {

}
```

```
public bool Unlock(int num, int user) {

}

public bool Upgrade(int num, int user) {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* LockingTree obj = new LockingTree(parent);
* bool param_1 = obj.Lock(num,user);
* bool param_2 = obj.Unlock(num,user);
* bool param_3 = obj.Upgrade(num,user);
*/
```

C:

```
typedef struct {

} LockingTree;


LockingTree* lockingTreeCreate(int* parent, int parentSize) {

}

bool lockingTreeLock(LockingTree* obj, int num, int user) {

}

bool lockingTreeUnlock(LockingTree* obj, int num, int user) {

}

bool lockingTreeUpgrade(LockingTree* obj, int num, int user) {

}
```

```
void lockingTreeFree(LockingTree* obj) {

}

/**
* Your LockingTree struct will be instantiated and called as such:
* LockingTree* obj = lockingTreeCreate(parent, parentSize);
* bool param_1 = lockingTreeLock(obj, num, user);

* bool param_2 = lockingTreeUnlock(obj, num, user);

* bool param_3 = lockingTreeUpgrade(obj, num, user);

* lockingTreeFree(obj);
*/
```

Go:

```
type LockingTree struct {

}


func Constructor(parent []int) LockingTree {

}


func (this *LockingTree) Lock(num int, user int) bool {

}


func (this *LockingTree) Unlock(num int, user int) bool {

}


func (this *LockingTree) Upgrade(num int, user int) bool {

}
```

```
/**
* Your LockingTree object will be instantiated and called as such:
* obj := Constructor(parent);
* param_1 := obj.Lock(num,user);
* param_2 := obj.Unlock(num,user);
* param_3 := obj.Upgrade(num,user);
*/
```

Kotlin:

```
class LockingTree(parent: IntArray) {

fun lock(num: Int, user: Int): Boolean {

}

fun unlock(num: Int, user: Int): Boolean {

}

fun upgrade(num: Int, user: Int): Boolean {

}

}

/**
* Your LockingTree object will be instantiated and called as such:
* var obj = LockingTree(parent)
* var param_1 = obj.lock(num,user)
* var param_2 = obj.unlock(num,user)
* var param_3 = obj.upgrade(num,user)
*/
```

Swift:

```
class LockingTree {

init(_ parent: [Int]) {
```

```
}

func lock(_ num: Int, _ user: Int) -> Bool {

}

func unlock(_ num: Int, _ user: Int) -> Bool {

}

func upgrade(_ num: Int, _ user: Int) -> Bool {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* let obj = LockingTree(parent)
* let ret_1: Bool = obj.lock(num, user)
* let ret_2: Bool = obj.unlock(num, user)
* let ret_3: Bool = obj.upgrade(num, user)
*/
```

Rust:

```
struct LockingTree {

}


/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl LockingTree {

fn new(parent: Vec<i32>) -> Self {

}

fn lock(&self, num: i32, user: i32) -> bool {
```

```
}

fn unlock(&self, num: i32, user: i32) -> bool {

}

fn upgrade(&self, num: i32, user: i32) -> bool {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* let obj = LockingTree::new(parent);
* let ret_1: bool = obj.lock(num, user);
* let ret_2: bool = obj.unlock(num, user);
* let ret_3: bool = obj.upgrade(num, user);
*/
```

Ruby:

```
class LockingTree

=begin
:type parent: Integer[]
=end
def initialize(parent)

end


=begin
:type num: Integer
:type user: Integer
:rtype: Boolean
=end
def lock(num, user)

end
```

```
=begin
:type num: Integer
:type user: Integer
:rtype: Boolean
=end
def unlock(num, user)

end


=begin
:type num: Integer
:type user: Integer
:rtype: Boolean
=end
def upgrade(num, user)

end



end

# Your LockingTree object will be instantiated and called as such:
# obj = LockingTree.new(parent)
# param_1 = obj.lock(num, user)
# param_2 = obj.unlock(num, user)
# param_3 = obj.upgrade(num, user)
```

PHP:

```
class LockingTree {
/**
* @param Integer[] $parent
*/
function __construct($parent) {

}

/**
* @param Integer $num
* @param Integer $user
* @return Boolean
```

```
*/
function lock($num, $user) {

}

/**
* @param Integer $num
* @param Integer $user
* @return Boolean
*/
function unlock($num, $user) {

}

/**
* @param Integer $num
* @param Integer $user
* @return Boolean
*/
function upgrade($num, $user) {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* $obj = LockingTree($parent);
* $ret_1 = $obj->lock($num, $user);
* $ret_2 = $obj->unlock($num, $user);
* $ret_3 = $obj->upgrade($num, $user);
*/
```

Dart:

```
class LockingTree {

LockingTree(List<int> parent) {

}

bool lock(int num, int user) {
```

```
}

bool unlock(int num, int user) {

}

bool upgrade(int num, int user) {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* LockingTree obj = LockingTree(parent);
* bool param1 = obj.lock(num,user);
* bool param2 = obj.unlock(num,user);
* bool param3 = obj.upgrade(num,user);
*/
```

Scala:

```
class LockingTree(_parent: Array[Int]) {

def lock(num: Int, user: Int): Boolean = {

}

def unlock(num: Int, user: Int): Boolean = {

}

def upgrade(num: Int, user: Int): Boolean = {

}

}

/**
* Your LockingTree object will be instantiated and called as such:
* val obj = new LockingTree(parent)
* val param_1 = obj.lock(num,user)
* val param_2 = obj.unlock(num,user)
```

```
* val param_3 = obj.upgrade(num,user)
*/
```

Elixir:

```
defmodule LockingTree do
@spec init_(parent :: [integer]) :: any
def init_(parent) do

end

@spec lock(num :: integer, user :: integer) :: boolean
def lock(num, user) do

end

@spec unlock(num :: integer, user :: integer) :: boolean
def unlock(num, user) do

end

@spec upgrade(num :: integer, user :: integer) :: boolean
def upgrade(num, user) do

end
end

# Your functions will be called as such:
# LockingTree.init_(parent)
# param_1 = LockingTree.lock(num, user)
# param_2 = LockingTree.unlock(num, user)
# param_3 = LockingTree.upgrade(num, user)

# LockingTree.init_ will be called before every test case, in which you can
do some necessary initializations.
```

Erlang:

```
-spec locking_tree_init_(Parent :: [integer()]) -> any().
locking_tree_init_(Parent) ->
.
```

```
-spec locking_tree_lock(Num :: integer(), User :: integer()) -> boolean().
locking_tree_lock(Num, User) ->
.

-spec locking_tree_unlock(Num :: integer(), User :: integer()) -> boolean().
locking_tree_unlock(Num, User) ->
.

-spec locking_tree_upgrade(Num :: integer(), User :: integer()) -> boolean().
locking_tree_upgrade(Num, User) ->
.


%% Your functions will be called as such:
%% locking_tree_init_(Parent),
%% Param_1 = locking_tree_lock(Num, User),
%% Param_2 = locking_tree_unlock(Num, User),
%% Param_3 = locking_tree_upgrade(Num, User),

%% locking_tree_init_ will be called before every test case, in which you can
do some necessary initializations.
```

Racket:

```
(define locking-tree%
(class object%
(super-new)

; parent : (listof exact-integer?)
(init-field
parent)

; lock : exact-integer? exact-integer? -> boolean?
(define/public (lock num user)
)
; unlock : exact-integer? exact-integer? -> boolean?
(define/public (unlock num user)
)
; upgrade : exact-integer? exact-integer? -> boolean?
(define/public (upgrade num user)
)))
```

```
;; Your locking-tree% object will be instantiated and called as such:
;; (define obj (new locking-tree% [parent parent]))
;; (define param_1 (send obj lock num user))
;; (define param_2 (send obj unlock num user))
;; (define param_3 (send obj upgrade num user))
```

Solutions

C++ Solution:

```
/*
* Problem: Operations on Tree
* Difficulty: Medium
* Tags: array, tree, hash, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class LockingTree {
public:
LockingTree(vector<int>& parent) {

}

bool lock(int num, int user) {

}

bool unlock(int num, int user) {

}

bool upgrade(int num, int user) {

}
};

/**
```

```
* Your LockingTree object will be instantiated and called as such:
* LockingTree* obj = new LockingTree(parent);
* bool param_1 = obj->lock(num,user);
* bool param_2 = obj->unlock(num,user);
* bool param_3 = obj->upgrade(num,user);
*/
```

Java Solution:

```
/**
* Problem: Operations on Tree
* Difficulty: Medium
* Tags: array, tree, hash, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class LockingTree {

public LockingTree(int[] parent) {

}

public boolean lock(int num, int user) {

}

public boolean unlock(int num, int user) {

}

public boolean upgrade(int num, int user) {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* LockingTree obj = new LockingTree(parent);
```

```
* boolean param_1 = obj.lock(num,user);
* boolean param_2 = obj.unlock(num,user);
* boolean param_3 = obj.upgrade(num,user);
*/
```

Python3 Solution:

```
"""
Problem: Operations on Tree
Difficulty: Medium
Tags: array, tree, hash, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

class LockingTree:

def __init__(self, parent: List[int]):


def lock(self, num: int, user: int) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class LockingTree(object):

def __init__(self, parent):
"""
:type parent: List[int]
"""


def lock(self, num, user):
"""
:type num: int
:type user: int
:rtype: bool
```

```
"""


def unlock(self, num, user):
"""
:type num: int
:type user: int
:rtype: bool
"""


def upgrade(self, num, user):
"""
:type num: int
:type user: int
:rtype: bool
"""



# Your LockingTree object will be instantiated and called as such:
# obj = LockingTree(parent)
# param_1 = obj.lock(num,user)
# param_2 = obj.unlock(num,user)
# param_3 = obj.upgrade(num,user)
```

JavaScript Solution:

```
/**
* Problem: Operations on Tree
* Difficulty: Medium
* Tags: array, tree, hash, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* @param {number[]} parent
*/
```

```
var LockingTree = function(parent) {

};

/**
* @param {number} num
* @param {number} user
* @return {boolean}
*/
LockingTree.prototype.lock = function(num, user) {

};

/**
* @param {number} num
* @param {number} user
* @return {boolean}
*/
LockingTree.prototype.unlock = function(num, user) {

};

/**
* @param {number} num
* @param {number} user
* @return {boolean}
*/
LockingTree.prototype.upgrade = function(num, user) {

};

/**
* Your LockingTree object will be instantiated and called as such:
* var obj = new LockingTree(parent)
* var param_1 = obj.lock(num,user)
* var param_2 = obj.unlock(num,user)
* var param_3 = obj.upgrade(num,user)
*/
```

TypeScript Solution:

```
/**
* Problem: Operations on Tree
* Difficulty: Medium
* Tags: array, tree, hash, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class LockingTree {
constructor(parent: number[]) {

}

lock(num: number, user: number): boolean {

}

unlock(num: number, user: number): boolean {

}

upgrade(num: number, user: number): boolean {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* var obj = new LockingTree(parent)
* var param_1 = obj.lock(num,user)
* var param_2 = obj.unlock(num,user)
* var param_3 = obj.upgrade(num,user)
*/
```

C# Solution:

```
/*
* Problem: Operations on Tree
* Difficulty: Medium
* Tags: array, tree, hash, search
```

```
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

public class LockingTree {

public LockingTree(int[] parent) {

}

public bool Lock(int num, int user) {

}

public bool Unlock(int num, int user) {

}

public bool Upgrade(int num, int user) {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* LockingTree obj = new LockingTree(parent);
* bool param_1 = obj.Lock(num,user);
* bool param_2 = obj.Unlock(num,user);
* bool param_3 = obj.Upgrade(num,user);
*/
```

C Solution:

```
/*
* Problem: Operations on Tree
* Difficulty: Medium
* Tags: array, tree, hash, search
*
* Approach: Use two pointers or sliding window technique
```

```
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/



typedef struct {

} LockingTree;


LockingTree* lockingTreeCreate(int* parent, int parentSize) {

}

bool lockingTreeLock(LockingTree* obj, int num, int user) {

}

bool lockingTreeUnlock(LockingTree* obj, int num, int user) {

}

bool lockingTreeUpgrade(LockingTree* obj, int num, int user) {

}

void lockingTreeFree(LockingTree* obj) {

}

/**
* Your LockingTree struct will be instantiated and called as such:
* LockingTree* obj = lockingTreeCreate(parent, parentSize);
* bool param_1 = lockingTreeLock(obj, num, user);

* bool param_2 = lockingTreeUnlock(obj, num, user);

* bool param_3 = lockingTreeUpgrade(obj, num, user);

* lockingTreeFree(obj);
```

```
*/
```

Go Solution:

```
// Problem: Operations on Tree
// Difficulty: Medium
// Tags: array, tree, hash, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

type LockingTree struct {

}


func Constructor(parent []int) LockingTree {

}


func (this *LockingTree) Lock(num int, user int) bool {

}


func (this *LockingTree) Unlock(num int, user int) bool {

}


func (this *LockingTree) Upgrade(num int, user int) bool {

}


/**
* Your LockingTree object will be instantiated and called as such:
* obj := Constructor(parent);
* param_1 := obj.Lock(num,user);
```

```
* param_2 := obj.Unlock(num,user);
* param_3 := obj.Upgrade(num,user);
*/
```

Kotlin Solution:

```
class LockingTree(parent: IntArray) {

fun lock(num: Int, user: Int): Boolean {

}

fun unlock(num: Int, user: Int): Boolean {

}

fun upgrade(num: Int, user: Int): Boolean {

}

}

/**
* Your LockingTree object will be instantiated and called as such:
* var obj = LockingTree(parent)
* var param_1 = obj.lock(num,user)
* var param_2 = obj.unlock(num,user)
* var param_3 = obj.upgrade(num,user)
*/
```

Swift Solution:

```
class LockingTree {

init(_ parent: [Int]) {

}

func lock(_ num: Int, _ user: Int) -> Bool {
```

```
}

func unlock(_ num: Int, _ user: Int) -> Bool {

}

func upgrade(_ num: Int, _ user: Int) -> Bool {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* let obj = LockingTree(parent)
* let ret_1: Bool = obj.lock(num, user)
* let ret_2: Bool = obj.unlock(num, user)
* let ret_3: Bool = obj.upgrade(num, user)
*/
```

Rust Solution:

```
// Problem: Operations on Tree
// Difficulty: Medium
// Tags: array, tree, hash, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

struct LockingTree {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl LockingTree {

fn new(parent: Vec<i32>) -> Self {
```

```
}

fn lock(&self, num: i32, user: i32) -> bool {

}

fn unlock(&self, num: i32, user: i32) -> bool {

}

fn upgrade(&self, num: i32, user: i32) -> bool {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* let obj = LockingTree::new(parent);
* let ret_1: bool = obj.lock(num, user);
* let ret_2: bool = obj.unlock(num, user);
* let ret_3: bool = obj.upgrade(num, user);
*/
```

Ruby Solution:

```
class LockingTree

=begin
:type parent: Integer[]
=end
def initialize(parent)

end


=begin
:type num: Integer
:type user: Integer
:rtype: Boolean
=end
```

```
def lock(num, user)

end


=begin
:type num: Integer
:type user: Integer
:rtype: Boolean
=end
def unlock(num, user)

end


=begin
:type num: Integer
:type user: Integer
:rtype: Boolean
=end
def upgrade(num, user)

end



end

# Your LockingTree object will be instantiated and called as such:
# obj = LockingTree.new(parent)
# param_1 = obj.lock(num, user)
# param_2 = obj.unlock(num, user)
# param_3 = obj.upgrade(num, user)
```

PHP Solution:

```
class LockingTree {
/**
* @param Integer[] $parent
*/
function __construct($parent) {
```

```
}

/**
* @param Integer $num
* @param Integer $user
* @return Boolean
*/
function lock($num, $user) {

}

/**
* @param Integer $num
* @param Integer $user
* @return Boolean
*/
function unlock($num, $user) {

}

/**
* @param Integer $num
* @param Integer $user
* @return Boolean
*/
function upgrade($num, $user) {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* $obj = LockingTree($parent);
* $ret_1 = $obj->lock($num, $user);
* $ret_2 = $obj->unlock($num, $user);
* $ret_3 = $obj->upgrade($num, $user);
*/
```

Dart Solution:

```
class LockingTree {

LockingTree(List<int> parent) {

}

bool lock(int num, int user) {

}

bool unlock(int num, int user) {

}

bool upgrade(int num, int user) {

}
}

/**
* Your LockingTree object will be instantiated and called as such:
* LockingTree obj = LockingTree(parent);
* bool param1 = obj.lock(num,user);
* bool param2 = obj.unlock(num,user);
* bool param3 = obj.upgrade(num,user);
*/
```

Scala Solution:

```
class LockingTree(_parent: Array[Int]) {

def lock(num: Int, user: Int): Boolean = {

}

def unlock(num: Int, user: Int): Boolean = {

}

def upgrade(num: Int, user: Int): Boolean = {

}
```

```
}

/**
* Your LockingTree object will be instantiated and called as such:
* val obj = new LockingTree(parent)
* val param_1 = obj.lock(num,user)
* val param_2 = obj.unlock(num,user)
* val param_3 = obj.upgrade(num,user)
*/
```

Elixir Solution:

```
defmodule LockingTree do
@spec init_(parent :: [integer]) :: any
def init_(parent) do

end

@spec lock(num :: integer, user :: integer) :: boolean
def lock(num, user) do

end

@spec unlock(num :: integer, user :: integer) :: boolean
def unlock(num, user) do

end

@spec upgrade(num :: integer, user :: integer) :: boolean
def upgrade(num, user) do

end
end

# Your functions will be called as such:
# LockingTree.init_(parent)
# param_1 = LockingTree.lock(num, user)
# param_2 = LockingTree.unlock(num, user)
# param_3 = LockingTree.upgrade(num, user)
```

```
# LockingTree.init_ will be called before every test case, in which you can
do some necessary initializations.
```

Erlang Solution:

```
-spec locking_tree_init_(Parent :: [integer()]) -> any().
locking_tree_init_(Parent) ->
.

-spec locking_tree_lock(Num :: integer(), User :: integer()) -> boolean().
locking_tree_lock(Num, User) ->
.

-spec locking_tree_unlock(Num :: integer(), User :: integer()) -> boolean().
locking_tree_unlock(Num, User) ->
.

-spec locking_tree_upgrade(Num :: integer(), User :: integer()) -> boolean().
locking_tree_upgrade(Num, User) ->
.


%% Your functions will be called as such:
%% locking_tree_init_(Parent),
%% Param_1 = locking_tree_lock(Num, User),
%% Param_2 = locking_tree_unlock(Num, User),
%% Param_3 = locking_tree_upgrade(Num, User),

%% locking_tree_init_ will be called before every test case, in which you can
do some necessary initializations.
```

Racket Solution:

```
(define locking-tree%
(class object%
(super-new)

; parent : (listof exact-integer?)
(init-field
parent)
```

```
; lock : exact-integer? exact-integer? -> boolean?
(define/public (lock num user)
)
; unlock : exact-integer? exact-integer? -> boolean?
(define/public (unlock num user)
)
; upgrade : exact-integer? exact-integer? -> boolean?
(define/public (upgrade num user)
)))

;; Your locking-tree% object will be instantiated and called as such:
;; (define obj (new locking-tree% [parent parent]))
;; (define param_1 (send obj lock num user))
;; (define param_2 (send obj unlock num user))
;; (define param_3 (send obj upgrade num user))
```