Problem 1483: Kth Ancestor of a Tree Node

Problem Information

Difficulty: Hard
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

i

th

node. The root of the tree is node

0

. Find the

k

th

ancestor of a given node.

The

k

th

ancestor of a tree node is the

k

th

node in the path from that node to the root node.

Implement the

TreeAncestor

class:

TreeAncestor(int n, int[] parent)

Initializes the object with the number of nodes in the tree and the parent array.

int getKthAncestor(int node, int k)

return the

k

th

ancestor of the given node

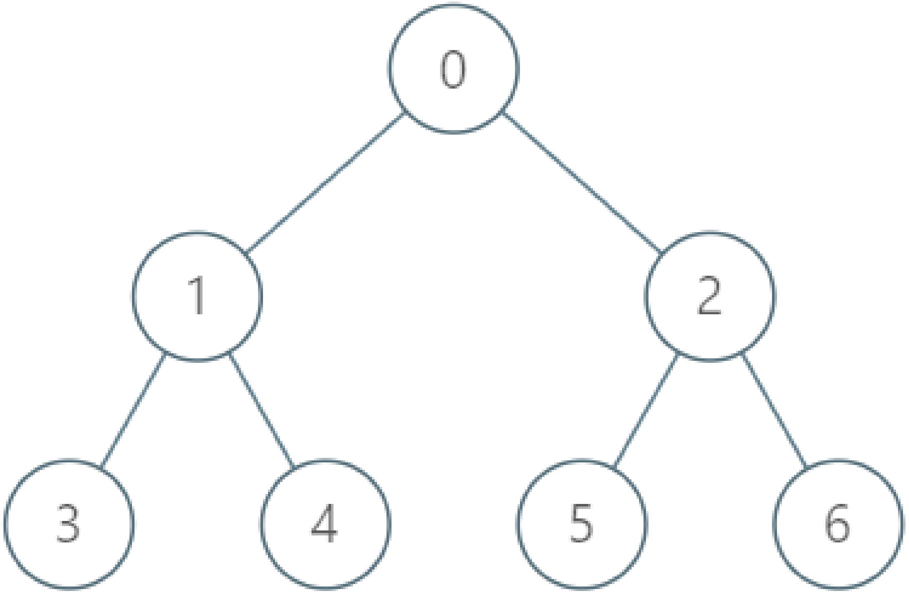
node

. If there is no such ancestor, return

-1

.

Example 1:



Input

["TreeAncestor", "getKthAncestor", "getKthAncestor", "getKthAncestor"] [[7, [-1, 0, 0, 1, 1, 2, 2]], [3, 1], [5, 2], [6, 3]]

Output

[null, 1, 0, -1]

Explanation

TreeAncestor treeAncestor = new TreeAncestor(7, [-1, 0, 0, 1, 1, 2, 2]);
treeAncestor.getKthAncestor(3, 1); // returns 1 which is the parent of 3
treeAncestor.getKthAncestor(5, 2); // returns 0 which is the grandparent of 5
treeAncestor.getKthAncestor(6, 3); // returns -1 because there is no such ancestor

Constraints:

1 <= k <= n <= 5 * 10

4

parent.length == n

parent[0] == -1

0 <= parent[i] < n

for all

0 < i < n

0 <= node < n

There will be at most

5 * 10

4

queries.

Code Snippets

C++:

```
class TreeAncestor {
public:
TreeAncestor(int n, vector<int>& parent) {

}

int getKthAncestor(int node, int k) {

}
};

/**
* Your TreeAncestor object will be instantiated and called as such:
* TreeAncestor* obj = new TreeAncestor(n, parent);
* int param_1 = obj->getKthAncestor(node,k);
*/
```

Java:

```
class TreeAncestor {

public TreeAncestor(int n, int[] parent) {

}

public int getKthAncestor(int node, int k) {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* TreeAncestor obj = new TreeAncestor(n, parent);
* int param_1 = obj.getKthAncestor(node,k);
*/
```

Python3:

```
class TreeAncestor:

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

```
def getKthAncestor(self, node: int, k: int) -> int:



# Your TreeAncestor object will be instantiated and called as such:
# obj = TreeAncestor(n, parent)
# param_1 = obj.getKthAncestor(node,k)
```

Python:

```
class TreeAncestor(object):

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


def getKthAncestor(self, node, k):
"""
:type node: int
:type k: int
:rtype: int
"""



# Your TreeAncestor object will be instantiated and called as such:
# obj = TreeAncestor(n, parent)
# param_1 = obj.getKthAncestor(node,k)
```

JavaScript:

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

};
```

```
/**
* @param {number} node
* @param {number} k
* @return {number}
*/
TreeAncestor.prototype.getKthAncestor = function(node, k) {

};

/**
* Your TreeAncestor object will be instantiated and called as such:
* var obj = new TreeAncestor(n, parent)
* var param_1 = obj.getKthAncestor(node,k)
*/
```

TypeScript:

```
class TreeAncestor {
constructor(n: number, parent: number[]) {

}

getKthAncestor(node: number, k: number): number {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* var obj = new TreeAncestor(n, parent)
* var param_1 = obj.getKthAncestor(node,k)
*/
```

C#:

```
public class TreeAncestor {

public TreeAncestor(int n, int[] parent) {

}

public int GetKthAncestor(int node, int k) {
}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* TreeAncestor obj = new TreeAncestor(n, parent);
* int param_1 = obj.GetKthAncestor(node,k);
*/
```

C:

```
typedef struct {

} TreeAncestor;


TreeAncestor* treeAncestorCreate(int n, int* parent, int parentSize) {

}

int treeAncestorGetKthAncestor(TreeAncestor* obj, int node, int k) {

}

void treeAncestorFree(TreeAncestor* obj) {

}

/**
* Your TreeAncestor struct will be instantiated and called as such:
* TreeAncestor* obj = treeAncestorCreate(n, parent, parentSize);
* int param_1 = treeAncestorGetKthAncestor(obj, node, k);

* treeAncestorFree(obj);
*/
```

Go:

```
type TreeAncestor struct {

}


func Constructor(n int, parent []int) TreeAncestor {

}


func (this *TreeAncestor) GetKthAncestor(node int, k int) int {

}


/**
* Your TreeAncestor object will be instantiated and called as such:
* obj := Constructor(n, parent);
* param_1 := obj.GetKthAncestor(node,k);
*/
```

Kotlin:

```
class TreeAncestor(n: Int, parent: IntArray) {

fun getKthAncestor(node: Int, k: Int): Int {

}

}

/**
* Your TreeAncestor object will be instantiated and called as such:
* var obj = TreeAncestor(n, parent)
* var param_1 = obj.getKthAncestor(node,k)
*/
```

Swift:

```

class TreeAncestor {

init(_ n: Int, _ parent: [Int]) {
}

func getKthAncestor(_ node: Int, _ k: Int) -> Int {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* let obj = TreeAncestor(n, parent)
* let ret_1: Int = obj.getKthAncestor(node, k)
*/
```

Rust:

```
struct TreeAncestor {

}


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

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

}

fn get_kth_ancestor(&self, node: i32, k: i32) -> i32 {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* let obj = TreeAncestor::new(n, parent);
* let ret_1: i32 = obj.get_kth_ancestor(node, k);
*/
```

Ruby:

```
class TreeAncestor

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

end


=begin
:type node: Integer
:type k: Integer
:rtype: Integer
=end
def get_kth_ancestor(node, k)

end


end

# Your TreeAncestor object will be instantiated and called as such:
# obj = TreeAncestor.new(n, parent)
# param_1 = obj.get_kth_ancestor(node, k)
```

PHP:

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

}

/**
* @param Integer $node
* @param Integer $k
* @return Integer
*/
function getKthAncestor($node, $k) {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* $obj = TreeAncestor($n, $parent);
* $ret_1 = $obj->getKthAncestor($node, $k);
*/
```

Dart:

```
class TreeAncestor {

TreeAncestor(int n, List<int> parent) {

}

int getKthAncestor(int node, int k) {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* TreeAncestor obj = TreeAncestor(n, parent);
* int param1 = obj.getKthAncestor(node,k);
*/
```

Scala:

```
class TreeAncestor(_n: Int, _parent: Array[Int]) {

def getKthAncestor(node: Int, k: Int): Int = {

}

}
```

```
/**
* Your TreeAncestor object will be instantiated and called as such:
* val obj = new TreeAncestor(n, parent)
* val param_1 = obj.getKthAncestor(node,k)
*/
```

Elixir:

```
defmodule TreeAncestor do
@spec init_(n :: integer, parent :: [integer]) :: any
def init_(n, parent) do

end

@spec get_kth_ancestor(node :: integer, k :: integer) :: integer
def get_kth_ancestor(node, k) do

end
end

# Your functions will be called as such:
# TreeAncestor.init_(n, parent)
# param_1 = TreeAncestor.get_kth_ancestor(node, k)

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

Erlang:

```
-spec tree_ancestor_init_(N :: integer(), Parent :: [integer()]) -> any().
tree_ancestor_init_(N, Parent) ->
.

-spec tree_ancestor_get_kth_ancestor(Node :: integer(), K :: integer()) ->
integer().
tree_ancestor_get_kth_ancestor(Node, K) ->
.



%% Your functions will be called as such:
%% tree_ancestor_init_(N, Parent),
%% Param_1 = tree_ancestor_get_kth_ancestor(Node, K),

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

Racket:

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

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

; get-kth-ancestor : exact-integer? exact-integer? -> exact-integer?
(define/public (get-kth-ancestor node k)
)))

;; Your tree-ancestor% object will be instantiated and called as such:
;; (define obj (new tree-ancestor% [n n] [parent parent]))
;; (define param_1 (send obj get-kth-ancestor node k))
```

Solutions

C++ Solution:

```
/*
* Problem: Kth Ancestor of a Tree Node
* Difficulty: Hard
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class TreeAncestor {
```

```
public:
TreeAncestor(int n, vector<int>& parent) {

}

int getKthAncestor(int node, int k) {

}
};

/**
* Your TreeAncestor object will be instantiated and called as such:
* TreeAncestor* obj = new TreeAncestor(n, parent);
* int param_1 = obj->getKthAncestor(node,k);
*/
```

Java Solution:

```
/**
* Problem: Kth Ancestor of a Tree Node
* Difficulty: Hard
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class TreeAncestor {

public TreeAncestor(int n, int[] parent) {

}

public int getKthAncestor(int node, int k) {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
```

```
* TreeAncestor obj = new TreeAncestor(n, parent);
* int param_1 = obj.getKthAncestor(node,k);
*/
```

Python3 Solution:

```
"""
Problem: Kth Ancestor of a Tree Node
Difficulty: Hard
Tags: array, tree, dp, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class TreeAncestor:

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



def getKthAncestor(self, node: int, k: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class TreeAncestor(object):

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



def getKthAncestor(self, node, k):
"""
:type node: int
:type k: int
:rtype: int
```

```
"""



# Your TreeAncestor object will be instantiated and called as such:
# obj = TreeAncestor(n, parent)
# param_1 = obj.getKthAncestor(node,k)
```

JavaScript Solution:

```
/**
* Problem: Kth Ancestor of a Tree Node
* Difficulty: Hard
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number} n
* @param {number[]} parent
*/
var TreeAncestor = function(n, parent) {

};

/**
* @param {number} node
* @param {number} k
* @return {number}
*/
TreeAncestor.prototype.getKthAncestor = function(node, k) {

};

/**
* Your TreeAncestor object will be instantiated and called as such:
* var obj = new TreeAncestor(n, parent)
* var param_1 = obj.getKthAncestor(node,k)
```

```

*/
```

TypeScript Solution:

```
/**
* Problem: Kth Ancestor of a Tree Node
* Difficulty: Hard
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class TreeAncestor {
constructor(n: number, parent: number[]) {

}

getKthAncestor(node: number, k: number): number {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* var obj = new TreeAncestor(n, parent)
* var param_1 = obj.getKthAncestor(node,k)
*/
```

C# Solution:

```
/*
* Problem: Kth Ancestor of a Tree Node
* Difficulty: Hard
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/
```

```
public class TreeAncestor {

public TreeAncestor(int n, int[] parent) {

}

public int GetKthAncestor(int node, int k) {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* TreeAncestor obj = new TreeAncestor(n, parent);
* int param_1 = obj.GetKthAncestor(node,k);
*/
```

C Solution:

```
/*
* Problem: Kth Ancestor of a Tree Node
* Difficulty: Hard
* Tags: array, tree, dp, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/




typedef struct {

} TreeAncestor;


TreeAncestor* treeAncestorCreate(int n, int* parent, int parentSize) {

}
```

```
int treeAncestorGetKthAncestor(TreeAncestor* obj, int node, int k) {

}

void treeAncestorFree(TreeAncestor* obj) {

}

/**
* Your TreeAncestor struct will be instantiated and called as such:
* TreeAncestor* obj = treeAncestorCreate(n, parent, parentSize);
* int param_1 = treeAncestorGetKthAncestor(obj, node, k);

* treeAncestorFree(obj);
*/
```

Go Solution:

```
// Problem: Kth Ancestor of a Tree Node
// Difficulty: Hard
// Tags: array, tree, dp, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

type TreeAncestor struct {

}


func Constructor(n int, parent []int) TreeAncestor {

}


func (this *TreeAncestor) GetKthAncestor(node int, k int) int {

}
```

```
/**
* Your TreeAncestor object will be instantiated and called as such:
* obj := Constructor(n, parent);
* param_1 := obj.GetKthAncestor(node,k);
*/
```

Kotlin Solution:

```
class TreeAncestor(n: Int, parent: IntArray) {

fun getKthAncestor(node: Int, k: Int): Int {

}

}

/**
* Your TreeAncestor object will be instantiated and called as such:
* var obj = TreeAncestor(n, parent)
* var param_1 = obj.getKthAncestor(node,k)
*/
```

Swift Solution:

```
class TreeAncestor {

init(_ n: Int, _ parent: [Int]) {

}

func getKthAncestor(_ node: Int, _ k: Int) -> Int {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* let obj = TreeAncestor(n, parent)
* let ret_1: Int = obj.getKthAncestor(node, k)
```

```

*/
```

Rust Solution:

```
// Problem: Kth Ancestor of a Tree Node
// Difficulty: Hard
// Tags: array, tree, dp, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

struct TreeAncestor {

}



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

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

}

fn get_kth_ancestor(&self, node: i32, k: i32) -> i32 {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* let obj = TreeAncestor::new(n, parent);
* let ret_1: i32 = obj.get_kth_ancestor(node, k);
*/
```

Ruby Solution:

```
class TreeAncestor

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

end



=begin
:type node: Integer
:type k: Integer
:rtype: Integer
=end
def get_kth_ancestor(node, k)

end



end

# Your TreeAncestor object will be instantiated and called as such:
# obj = TreeAncestor.new(n, parent)
# param_1 = obj.get_kth_ancestor(node, k)
```

PHP Solution:

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

}

/**
* @param Integer $node
* @param Integer $k
* @return Integer
*/
function getKthAncestor($node, $k) {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* $obj = TreeAncestor($n, $parent);
* $ret_1 = $obj->getKthAncestor($node, $k);
*/
```

Dart Solution:

```
class TreeAncestor {

TreeAncestor(int n, List<int> parent) {

}

int getKthAncestor(int node, int k) {

}
}

/**
* Your TreeAncestor object will be instantiated and called as such:
* TreeAncestor obj = TreeAncestor(n, parent);
* int param1 = obj.getKthAncestor(node,k);
*/
```

Scala Solution:

```
class TreeAncestor(_n: Int, _parent: Array[Int]) {

def getKthAncestor(node: Int, k: Int): Int = {

}

}
```

```
/**
* Your TreeAncestor object will be instantiated and called as such:
* val obj = new TreeAncestor(n, parent)
* val param_1 = obj.getKthAncestor(node,k)
*/
```

Elixir Solution:

```
defmodule TreeAncestor do
@spec init_(n :: integer, parent :: [integer]) :: any
def init_(n, parent) do

end

@spec get_kth_ancestor(node :: integer, k :: integer) :: integer
def get_kth_ancestor(node, k) do

end
end

# Your functions will be called as such:
# TreeAncestor.init_(n, parent)
# param_1 = TreeAncestor.get_kth_ancestor(node, k)

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

Erlang Solution:

```
-spec tree_ancestor_init_(N :: integer(), Parent :: [integer()]) -> any().
tree_ancestor_init_(N, Parent) ->
.

-spec tree_ancestor_get_kth_ancestor(Node :: integer(), K :: integer()) ->
integer().
tree_ancestor_get_kth_ancestor(Node, K) ->
.


%% Your functions will be called as such:
%% tree_ancestor_init_(N, Parent),
%% Param_1 = tree_ancestor_get_kth_ancestor(Node, K),

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

Racket Solution:

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

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

; get-kth-ancestor : exact-integer? exact-integer? -> exact-integer?
(define/public (get-kth-ancestor node k)
)))

;; Your tree-ancestor% object will be instantiated and called as such:
;; (define obj (new tree-ancestor% [n n] [parent parent]))
;; (define param_1 (send obj get-kth-ancestor node k))
```