Problem 1586: Binary Search Tree Iterator II

Problem Information

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

Problem Description

Implement the

BSTIterator

class that represents an iterator over the

in-order traversal

of a binary search tree (BST):

BSTIterator(TreeNode root)

Initializes an object of the

BSTIterator

class. The

root

of the BST is given as part of the constructor. The pointer should be initialized to a non-existent number smaller than any element in the BST.

boolean hasNext()

Returns

true

if there exists a number in the traversal to the right of the pointer, otherwise returns

false

.

int next()

Moves the pointer to the right, then returns the number at the pointer.

boolean hasPrev()

Returns

true

if there exists a number in the traversal to the left of the pointer, otherwise returns

false

.

int prev()

Moves the pointer to the left, then returns the number at the pointer.

Notice that by initializing the pointer to a non-existent smallest number, the first call to

next()

will return the smallest element in the BST.

You may assume that

next()

and

prev()

calls will always be valid. That is, there will be at least a next/previous number in the in-order traversal when

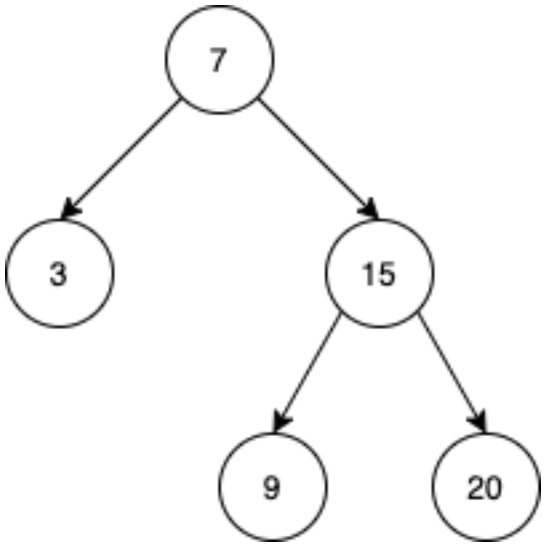
next()

/

prev()

is called.

Example 1:



Input

["BSTIterator", "next", "next", "prev", "next", "hasNext", "next", "next", "next", "hasNext", "hasPrev", "prev", "prev"] [[[7, 3, 15, null, null, 9, 20]], [null], [null], [null], [null], [null], [null], [null], [null], [null], [null], [null], [null]]

Output

[null, 3, 7, 3, 7, true, 9, 15, 20, false, true, 15, 9]

Explanation

// The underlined element is where the pointer currently is. BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]); // state is

[3, 7, 9, 15, 20] bSTIterator.next(); // state becomes [

3

, 7, 9, 15, 20], return 3 bSTIterator.next(); // state becomes [3,

7

, 9, 15, 20], return 7 bSTIterator.prev(); // state becomes [

3

, 7, 9, 15, 20], return 3 bSTIterator.next(); // state becomes [3,

7

, 9, 15, 20], return 7 bSTIterator.hasNext(); // return true bSTIterator.next(); // state becomes [3, 7,

9

, 15, 20], return 9 bSTIterator.next(); // state becomes [3, 7, 9,

15

, 20], return 15 bSTIterator.next(); // state becomes [3, 7, 9, 15,

20

], return 20 bSTIterator.hasNext(); // return false bSTIterator.hasPrev(); // return true bSTIterator.prev(); // state becomes [3, 7, 9,

15

, 20], return 15 bSTIterator.prev(); // state becomes [3, 7,

9

, 15, 20], return 9

Constraints:

The number of nodes in the tree is in the range

[1, 10

5

]

.

0 <= Node.val <= 10

6

At most

10

5

calls will be made to

hasNext

,

next

,

hasPrev

, and

prev

.

Follow up:

Could you solve the problem without precalculating the values of the tree?

Code Snippets

C++:

```
/**
* Definition for a binary tree node.
* struct TreeNode {
* int val;
* TreeNode *left;
* TreeNode *right;
* TreeNode() : val(0), left(nullptr), right(nullptr) {}
* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
* TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
* };
*/
class BSTIterator {
public:
BSTIterator(TreeNode* root) {

}

bool hasNext() {

}

int next() {

}
```

```
bool hasPrev() {

}

int prev() {

}
};

/**
* Your BSTIterator object will be instantiated and called as such:
* BSTIterator* obj = new BSTIterator(root);
* bool param_1 = obj->hasNext();
* int param_2 = obj->next();
* bool param_3 = obj->hasPrev();
* int param_4 = obj->prev();
*/
```

Java:

```
/**
* Definition for a binary tree node.
* public class TreeNode {
* int val;
* TreeNode left;
* TreeNode right;
* TreeNode() {}
* TreeNode(int val) { this.val = val; }
* TreeNode(int val, TreeNode left, TreeNode right) {
* this.val = val;
* this.left = left;
* this.right = right;
* }
* }
*/
class BSTIterator {

public BSTIterator(TreeNode root) {

}

public boolean hasNext() {
```

```
}

public int next() {

}

public boolean hasPrev() {

}

public int prev() {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* BSTIterator obj = new BSTIterator(root);
* boolean param_1 = obj.hasNext();
* int param_2 = obj.next();
* boolean param_3 = obj.hasPrev();
* int param_4 = obj.prev();
*/
```

Python3:

```
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class BSTIterator:

def __init__(self, root: Optional[TreeNode]):


def hasNext(self) -> bool:


def next(self) -> int:
```

```
def hasPrev(self) -> bool:


def prev(self) -> int:



# Your BSTIterator object will be instantiated and called as such:
# obj = BSTIterator(root)
# param_1 = obj.hasNext()
# param_2 = obj.next()
# param_3 = obj.hasPrev()
# param_4 = obj.prev()
```

Python:

```
# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class BSTIterator(object):

def __init__(self, root):
"""
:type root: Optional[TreeNode]
"""


def hasNext(self):
"""
:rtype: bool
"""


def next(self):
"""
:rtype: int
"""
```

```
def hasPrev(self):
"""
:rtype: bool
"""


def prev(self):
"""
:rtype: int
"""



# Your BSTIterator object will be instantiated and called as such:
# obj = BSTIterator(root)
# param_1 = obj.hasNext()
# param_2 = obj.next()
# param_3 = obj.hasPrev()
# param_4 = obj.prev()
```

JavaScript:

```
/**
* Definition for a binary tree node.
* function TreeNode(val, left, right) {
* this.val = (val===undefined ? 0 : val)
* this.left = (left===undefined ? null : left)
* this.right = (right===undefined ? null : right)
* }
*/
/**
* @param {TreeNode} root
*/
var BSTIterator = function(root) {

};

/**
* @return {boolean}
*/
```

```
BSTIterator.prototype.hasNext = function() {

};

/**
* @return {number}
*/
BSTIterator.prototype.next = function() {

};

/**
* @return {boolean}
*/
BSTIterator.prototype.hasPrev = function() {

};

/**
* @return {number}
*/
BSTIterator.prototype.prev = function() {

};

/**
* Your BSTIterator object will be instantiated and called as such:
* var obj = new BSTIterator(root)
* var param_1 = obj.hasNext()
* var param_2 = obj.next()
* var param_3 = obj.hasPrev()
* var param_4 = obj.prev()
*/
```

TypeScript:

```
/**
* Definition for a binary tree node.
* class TreeNode {
* val: number
* left: TreeNode | null
* right: TreeNode | null
```

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

class BSTIterator {
constructor(root: TreeNode | null) {

}

hasNext(): boolean {

}

next(): number {

}

hasPrev(): boolean {

}

prev(): number {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* var obj = new BSTIterator(root)
* var param_1 = obj.hasNext()
* var param_2 = obj.next()
* var param_3 = obj.hasPrev()
* var param_4 = obj.prev()
*/
```

C#:

```
/**
* Definition for a binary tree node.
* public class TreeNode {
* public int val;
* public TreeNode left;
* public TreeNode right;
* public TreeNode(int val=0, TreeNode left=null, TreeNode right=null) {
* this.val = val;
* this.left = left;
* this.right = right;
* }
* }
*/
public class BSTIterator {

public BSTIterator(TreeNode root) {

}

public bool HasNext() {

}

public int Next() {

}

public bool HasPrev() {

}

public int Prev() {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* BSTIterator obj = new BSTIterator(root);
* bool param_1 = obj.HasNext();
* int param_2 = obj.Next();
* bool param_3 = obj.HasPrev();
* int param_4 = obj.Prev();
```

```
*/
```

C:

```
/**
* Definition for a binary tree node.
* struct TreeNode {
* int val;
* struct TreeNode *left;
* struct TreeNode *right;
* };
*/



typedef struct {

} BSTIterator;


BSTIterator* bSTIteratorCreate(struct TreeNode* root) {

}

bool bSTIteratorHasNext(BSTIterator* obj) {

}

int bSTIteratorNext(BSTIterator* obj) {

}

bool bSTIteratorHasPrev(BSTIterator* obj) {

}

int bSTIteratorPrev(BSTIterator* obj) {

}

void bSTIteratorFree(BSTIterator* obj) {
```

```
}

/**
* Your BSTIterator struct will be instantiated and called as such:
* BSTIterator* obj = bSTIteratorCreate(root);
* bool param_1 = bSTIteratorHasNext(obj);

* int param_2 = bSTIteratorNext(obj);

* bool param_3 = bSTIteratorHasPrev(obj);

* int param_4 = bSTIteratorPrev(obj);

* bSTIteratorFree(obj);
*/
```

Go:

```
/**
* Definition for a binary tree node.
* type TreeNode struct {
* Val int
* Left *TreeNode
* Right *TreeNode
* }
*/
type BSTIterator struct {

}


func Constructor(root *TreeNode) BSTIterator {

}


func (this *BSTIterator) HasNext() bool {

}


func (this *BSTIterator) Next() int {
```

```
}


func (this *BSTIterator) HasPrev() bool {

}


func (this *BSTIterator) Prev() int {

}


/**
* Your BSTIterator object will be instantiated and called as such:
* obj := Constructor(root);
* param_1 := obj.HasNext();
* param_2 := obj.Next();
* param_3 := obj.HasPrev();
* param_4 := obj.Prev();
*/
```

Kotlin:

```
/**
* Example:
* var ti = TreeNode(5)
* var v = ti.`val`
* Definition for a binary tree node.
* class TreeNode(var `val`: Int) {
* var left: TreeNode? = null
* var right: TreeNode? = null
* }
*/
class BSTIterator(root: TreeNode?) {

fun hasNext(): Boolean {

}

fun next(): Int {
```

```
}

fun hasPrev(): Boolean {

}

fun prev(): Int {

}

}

/**
* Your BSTIterator object will be instantiated and called as such:
* var obj = BSTIterator(root)
* var param_1 = obj.hasNext()
* var param_2 = obj.next()
* var param_3 = obj.hasPrev()
* var param_4 = obj.prev()
*/
```

Swift:

```
/**
* Definition for a binary tree node.
* public class TreeNode {
* public var val: Int
* public var left: TreeNode?
* public var right: TreeNode?
* public init() { self.val = 0; self.left = nil; self.right = nil; }
* public init(_ val: Int) { self.val = val; self.left = nil; self.right =
nil; }
* public init(_ val: Int, _ left: TreeNode?, _ right: TreeNode?) {
* self.val = val
* self.left = left
* self.right = right
* }
* }
*/

class BSTIterator {
```

```
init(_ root: TreeNode?) {

}

func hasNext() -> Bool {

}

func next() -> Int {

}

func hasPrev() -> Bool {

}

func prev() -> Int {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* let obj = BSTIterator(root)
* let ret_1: Bool = obj.hasNext()
* let ret_2: Int = obj.next()
* let ret_3: Bool = obj.hasPrev()
* let ret_4: Int = obj.prev()
*/
```

Rust:

```
// Definition for a binary tree node.
// #[derive(Debug, PartialEq, Eq)]
// pub struct TreeNode {
// pub val: i32,
// pub left: Option<Rc<RefCell<TreeNode>>>,
// pub right: Option<Rc<RefCell<TreeNode>>>,
// }
//
// impl TreeNode {
```

```
// #[inline]
// pub fn new(val: i32) -> Self {
// TreeNode {
// val,
// left: None,
// right: None
// }
// }
// }
struct BSTIterator {

}


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

fn new(root: Option<Rc<RefCell<TreeNode>>>) -> Self {

}

fn has_next(&self) -> bool {

}

fn next(&self) -> i32 {

}

fn has_prev(&self) -> bool {

}

fn prev(&self) -> i32 {

}
}

/**
```

```
* Your BSTIterator object will be instantiated and called as such:
* let obj = BSTIterator::new(root);
* let ret_1: bool = obj.has_next();
* let ret_2: i32 = obj.next();
* let ret_3: bool = obj.has_prev();
* let ret_4: i32 = obj.prev();
*/
```

Ruby:

```
# Definition for a binary tree node.
# class TreeNode
# attr_accessor :val, :left, :right
# def initialize(val = 0, left = nil, right = nil)
# @val = val
# @left = left
# @right = right
# end
# end
class BSTIterator

=begin
:type root: TreeNode
=end
def initialize(root)

end


=begin
:rtype: Boolean
=end
def has_next()

end


=begin
:rtype: Integer
=end
def next()
```

```
end


=begin
:rtype: Boolean
=end
def has_prev()

end


=begin
:rtype: Integer
=end
def prev()

end


end

# Your BSTIterator object will be instantiated and called as such:
# obj = BSTIterator.new(root)
# param_1 = obj.has_next()
# param_2 = obj.next()
# param_3 = obj.has_prev()
# param_4 = obj.prev()
```

PHP:

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

```
*/
class BSTIterator {
/**
* @param TreeNode $root
*/
function __construct($root) {

}

/**
* @return Boolean
*/
function hasNext() {

}

/**
* @return Integer
*/
function next() {

}

/**
* @return Boolean
*/
function hasPrev() {

}

/**
* @return Integer
*/
function prev() {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* $obj = BSTIterator($root);
* $ret_1 = $obj->hasNext();
```

```
* $ret_2 = $obj->next();
* $ret_3 = $obj->hasPrev();
* $ret_4 = $obj->prev();
*/
```

Dart:

```
/**
* Definition for a binary tree node.
* class TreeNode {
* int val;
* TreeNode? left;
* TreeNode? right;
* TreeNode([this.val = 0, this.left, this.right]);
* }
*/
class BSTIterator {

BSTIterator(TreeNode? root) {

}

bool hasNext() {

}

int next() {

}

bool hasPrev() {

}

int prev() {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* BSTIterator obj = BSTIterator(root);
```

```
* bool param1 = obj.hasNext();
* int param2 = obj.next();
* bool param3 = obj.hasPrev();
* int param4 = obj.prev();
*/
```

Scala:

```
/**
* Definition for a binary tree node.
* class TreeNode(_value: Int = 0, _left: TreeNode = null, _right: TreeNode =
null) {
* var value: Int = _value
* var left: TreeNode = _left
* var right: TreeNode = _right
* }
*/
class BSTIterator(_root: TreeNode) {

def hasNext(): Boolean = {

}

def next(): Int = {

}

def hasPrev(): Boolean = {

}

def prev(): Int = {

}

}

/**
* Your BSTIterator object will be instantiated and called as such:
* val obj = new BSTIterator(root)
* val param_1 = obj.hasNext()
* val param_2 = obj.next()
```

```
* val param_3 = obj.hasPrev()
* val param_4 = obj.prev()
*/
```

Elixir:

```
# Definition for a binary tree node.
#
# defmodule TreeNode do
# @type t :: %__MODULE__{
# val: integer,
# left: TreeNode.t() | nil,
# right: TreeNode.t() | nil
# }
# defstruct val: 0, left: nil, right: nil
# end

defmodule BSTIterator do
@spec init_(root :: TreeNode.t | nil) :: any
def init_(root) do

end

@spec has_next() :: boolean
def has_next() do

end

@spec next() :: integer
def next() do

end

@spec has_prev() :: boolean
def has_prev() do

end

@spec prev() :: integer
def prev() do

end
```

```
end

# Your functions will be called as such:
# BSTIterator.init_(root)
# param_1 = BSTIterator.has_next()
# param_2 = BSTIterator.next()
# param_3 = BSTIterator.has_prev()
# param_4 = BSTIterator.prev()

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

Erlang:

```
%% Definition for a binary tree node.
%%
%% -record(tree_node, {val = 0 :: integer(),
%% left = null :: 'null' | #tree_node{},
%% right = null :: 'null' | #tree_node{}}).

-spec bst_iterator_init_(Root :: #tree_node{} | null) -> any().
bst_iterator_init_(Root) ->
.

-spec bst_iterator_has_next() -> boolean().
bst_iterator_has_next() ->
.

-spec bst_iterator_next() -> integer().
bst_iterator_next() ->
.

-spec bst_iterator_has_prev() -> boolean().
bst_iterator_has_prev() ->
.

-spec bst_iterator_prev() -> integer().
bst_iterator_prev() ->
.


%% Your functions will be called as such:
```

```
%% bst_iterator_init_(Root),
%% Param_1 = bst_iterator_has_next(),
%% Param_2 = bst_iterator_next(),
%% Param_3 = bst_iterator_has_prev(),
%% Param_4 = bst_iterator_prev(),

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

Racket:

```
; Definition for a binary tree node.
#|

; val : integer?
; left : (or/c tree-node? #f)
; right : (or/c tree-node? #f)
(struct tree-node
(val left right) #:mutable #:transparent)

; constructor
(define (make-tree-node [val 0])
(tree-node val #f #f))

|#

(define bst-iterator%
(class object%
(super-new)

; root : (or/c tree-node? #f)
(init-field
root)

; has-next : -> boolean?
(define/public (has-next)
)
; next : -> exact-integer?
(define/public (next)
)
; has-prev : -> boolean?
(define/public (has-prev)
```

```
)
; prev : -> exact-integer?
(define/public (prev)
)))

;; Your bst-iterator% object will be instantiated and called as such:
;; (define obj (new bst-iterator% [root root]))
;; (define param_1 (send obj has-next))
;; (define param_2 (send obj next))
;; (define param_3 (send obj has-prev))
;; (define param_4 (send obj prev))
```

Solutions

C++ Solution:

```
/*
* Problem: Binary Search Tree Iterator II
* Difficulty: Medium
* Tags: tree, search, 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 binary tree node.
* struct TreeNode {
* int val;
* TreeNode *left;
* TreeNode *right;
* TreeNode() : val(0), left(nullptr), right(nullptr) {
// TODO: Implement optimized solution
return 0;
}
* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {
// TODO: Implement optimized solution
return 0;
}
```

```
* TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {
// TODO: Implement optimized solution
return 0;
}
* };
*/
class BSTIterator {
public:
BSTIterator(TreeNode* root) {

}

bool hasNext() {

}

int next() {

}

bool hasPrev() {

}

int prev() {

}
};

/**
* Your BSTIterator object will be instantiated and called as such:
* BSTIterator* obj = new BSTIterator(root);
* bool param_1 = obj->hasNext();
* int param_2 = obj->next();
* bool param_3 = obj->hasPrev();
* int param_4 = obj->prev();
*/
```

Java Solution:

```
/**
* Problem: Binary Search Tree Iterator II
* Difficulty: Medium
* Tags: tree, search, 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 binary tree node.
* public class TreeNode {
* int val;
* TreeNode left;
* TreeNode right;
* TreeNode() {
// TODO: Implement optimized solution
return 0;
}
* TreeNode(int val) { this.val = val; }
* TreeNode(int val, TreeNode left, TreeNode right) {
* this.val = val;
* this.left = left;
* this.right = right;
* }
* }
*/
class BSTIterator {

public BSTIterator(TreeNode root) {

}

public boolean hasNext() {

}

public int next() {

}

public boolean hasPrev() {
```

```
}

public int prev() {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* BSTIterator obj = new BSTIterator(root);
* boolean param_1 = obj.hasNext();
* int param_2 = obj.next();
* boolean param_3 = obj.hasPrev();
* int param_4 = obj.prev();
*/
```

Python3 Solution:

```
"""
Problem: Binary Search Tree Iterator II
Difficulty: Medium
Tags: tree, search, 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 binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class BSTIterator:

def __init__(self, root: Optional[TreeNode]):


def hasNext(self) -> bool:
```

```
# TODO: Implement optimized solution
pass
```

Python Solution:

```
# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class BSTIterator(object):

def __init__(self, root):
"""
:type root: Optional[TreeNode]
"""


def hasNext(self):
"""
:rtype: bool
"""


def next(self):
"""
:rtype: int
"""


def hasPrev(self):
"""
:rtype: bool
"""


def prev(self):
"""
:rtype: int
"""
```

```
# Your BSTIterator object will be instantiated and called as such:
# obj = BSTIterator(root)
# param_1 = obj.hasNext()
# param_2 = obj.next()
# param_3 = obj.hasPrev()
# param_4 = obj.prev()
```

JavaScript Solution:

```
/**
* Problem: Binary Search Tree Iterator II
* Difficulty: Medium
* Tags: tree, search, 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 binary tree node.
* function TreeNode(val, left, right) {
* this.val = (val===undefined ? 0 : val)
* this.left = (left===undefined ? null : left)
* this.right = (right===undefined ? null : right)
* }
*/
/**
* @param {TreeNode} root
*/
var BSTIterator = function(root) {

};

/**
* @return {boolean}
*/
BSTIterator.prototype.hasNext = function() {
```

```
};

/**
* @return {number}
*/
BSTIterator.prototype.next = function() {

};

/**
* @return {boolean}
*/
BSTIterator.prototype.hasPrev = function() {

};

/**
* @return {number}
*/
BSTIterator.prototype.prev = function() {

};

/**
* Your BSTIterator object will be instantiated and called as such:
* var obj = new BSTIterator(root)
* var param_1 = obj.hasNext()
* var param_2 = obj.next()
* var param_3 = obj.hasPrev()
* var param_4 = obj.prev()
*/
```

TypeScript Solution:

```
/**
* Problem: Binary Search Tree Iterator II
* Difficulty: Medium
* Tags: tree, search, 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 binary tree node.
* class TreeNode {
* val: number
* left: TreeNode | null
* right: TreeNode | null
* constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null)
{
* this.val = (val===undefined ? 0 : val)
* this.left = (left===undefined ? null : left)
* this.right = (right===undefined ? null : right)
* }
* }
*/

class BSTIterator {
constructor(root: TreeNode | null) {

}

hasNext(): boolean {

}

next(): number {

}

hasPrev(): boolean {

}

prev(): number {

}
}

/**
```

```
* Your BSTIterator object will be instantiated and called as such:
* var obj = new BSTIterator(root)
* var param_1 = obj.hasNext()
* var param_2 = obj.next()
* var param_3 = obj.hasPrev()
* var param_4 = obj.prev()
*/
```

C# Solution:

```
/*
* Problem: Binary Search Tree Iterator II
* Difficulty: Medium
* Tags: tree, search, 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 binary tree node.
* public class TreeNode {
* public int val;
* public TreeNode left;
* public TreeNode right;
* public TreeNode(int val=0, TreeNode left=null, TreeNode right=null) {
* this.val = val;
* this.left = left;
* this.right = right;
* }
* }
*/
public class BSTIterator {

public BSTIterator(TreeNode root) {

}

public bool HasNext() {
```

```
}

public int Next() {

}

public bool HasPrev() {

}

public int Prev() {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* BSTIterator obj = new BSTIterator(root);
* bool param_1 = obj.HasNext();
* int param_2 = obj.Next();
* bool param_3 = obj.HasPrev();
* int param_4 = obj.Prev();
*/
```

C Solution:

```
/*
* Problem: Binary Search Tree Iterator II
* Difficulty: Medium
* Tags: tree, search, 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 binary tree node.
* struct TreeNode {
* int val;
* struct TreeNode *left;
```

```
* struct TreeNode *right;
* };
*/



typedef struct {

} BSTIterator;


BSTIterator* bSTIteratorCreate(struct TreeNode* root) {

}

bool bSTIteratorHasNext(BSTIterator* obj) {

}

int bSTIteratorNext(BSTIterator* obj) {

}

bool bSTIteratorHasPrev(BSTIterator* obj) {

}

int bSTIteratorPrev(BSTIterator* obj) {

}

void bSTIteratorFree(BSTIterator* obj) {

}

/**
* Your BSTIterator struct will be instantiated and called as such:
* BSTIterator* obj = bSTIteratorCreate(root);
* bool param_1 = bSTIteratorHasNext(obj);

* int param_2 = bSTIteratorNext(obj);
```

```
* bool param_3 = bSTIteratorHasPrev(obj);

* int param_4 = bSTIteratorPrev(obj);

* bSTIteratorFree(obj);
*/
```

Go Solution:

```
// Problem: Binary Search Tree Iterator II
// Difficulty: Medium
// Tags: tree, search, 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 binary tree node.
* type TreeNode struct {
* Val int
* Left *TreeNode
* Right *TreeNode
* }
*/
type BSTIterator struct {

}


func Constructor(root *TreeNode) BSTIterator {

}


func (this *BSTIterator) HasNext() bool {

}


func (this *BSTIterator) Next() int {
```

```
}


func (this *BSTIterator) HasPrev() bool {

}


func (this *BSTIterator) Prev() int {

}


/**
* Your BSTIterator object will be instantiated and called as such:
* obj := Constructor(root);
* param_1 := obj.HasNext();
* param_2 := obj.Next();
* param_3 := obj.HasPrev();
* param_4 := obj.Prev();
*/
```

Kotlin Solution:

```
/**
* Example:
* var ti = TreeNode(5)
* var v = ti.`val`
* Definition for a binary tree node.
* class TreeNode(var `val`: Int) {
* var left: TreeNode? = null
* var right: TreeNode? = null
* }
*/
class BSTIterator(root: TreeNode?) {

fun hasNext(): Boolean {

}
```

```
fun next(): Int {

}

fun hasPrev(): Boolean {

}

fun prev(): Int {

}

}

/**
* Your BSTIterator object will be instantiated and called as such:
* var obj = BSTIterator(root)
* var param_1 = obj.hasNext()
* var param_2 = obj.next()
* var param_3 = obj.hasPrev()
* var param_4 = obj.prev()
*/
```

Swift Solution:

```
/**
* Definition for a binary tree node.
* public class TreeNode {
* public var val: Int
* public var left: TreeNode?
* public var right: TreeNode?
* public init() { self.val = 0; self.left = nil; self.right = nil; }
* public init(_ val: Int) { self.val = val; self.left = nil; self.right =
nil; }
* public init(_ val: Int, _ left: TreeNode?, _ right: TreeNode?) {
* self.val = val
* self.left = left
* self.right = right
* }
* }
*/
```

```
class BSTIterator {

init(_ root: TreeNode?) {

}

func hasNext() -> Bool {

}

func next() -> Int {

}

func hasPrev() -> Bool {

}

func prev() -> Int {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* let obj = BSTIterator(root)
* let ret_1: Bool = obj.hasNext()
* let ret_2: Int = obj.next()
* let ret_3: Bool = obj.hasPrev()
* let ret_4: Int = obj.prev()
*/
```

Rust Solution:

```
// Problem: Binary Search Tree Iterator II
// Difficulty: Medium
// Tags: tree, search, 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 binary tree node.
// #[derive(Debug, PartialEq, Eq)]
// pub struct TreeNode {
// pub val: i32,
// pub left: Option<Rc<RefCell<TreeNode>>>,
// pub right: Option<Rc<RefCell<TreeNode>>>,
// }
//
// impl TreeNode {
// #[inline]
// pub fn new(val: i32) -> Self {
// TreeNode {
// val,
// left: None,
// right: None
// }
// }
// }
struct BSTIterator {

}


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

fn new(root: Option<Rc<RefCell<TreeNode>>>) -> Self {

}

fn has_next(&self) -> bool {

}

fn next(&self) -> i32 {

}
```

```
fn has_prev(&self) -> bool {

}

fn prev(&self) -> i32 {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* let obj = BSTIterator::new(root);
* let ret_1: bool = obj.has_next();
* let ret_2: i32 = obj.next();
* let ret_3: bool = obj.has_prev();
* let ret_4: i32 = obj.prev();
*/
```

Ruby Solution:

```
# Definition for a binary tree node.
# class TreeNode
# attr_accessor :val, :left, :right
# def initialize(val = 0, left = nil, right = nil)
# @val = val
# @left = left
# @right = right
# end
# end
class BSTIterator

=begin
:type root: TreeNode
=end
def initialize(root)

end


=begin
```

```
:rtype: Boolean
=end
def has_next()

end


=begin
:rtype: Integer
=end
def next()

end


=begin
:rtype: Boolean
=end
def has_prev()

end


=begin
:rtype: Integer
=end
def prev()

end


end

# Your BSTIterator object will be instantiated and called as such:
# obj = BSTIterator.new(root)
# param_1 = obj.has_next()
# param_2 = obj.next()
# param_3 = obj.has_prev()
# param_4 = obj.prev()
```

PHP Solution:

```
/**
* Definition for a binary tree node.
* class TreeNode {
* public $val = null;
* public $left = null;
* public $right = null;
* function __construct($val = 0, $left = null, $right = null) {
* $this->val = $val;
* $this->left = $left;
* $this->right = $right;
* }
* }
*/
class BSTIterator {
/**
* @param TreeNode $root
*/
function __construct($root) {

}

/**
* @return Boolean
*/
function hasNext() {

}

/**
* @return Integer
*/
function next() {

}

/**
* @return Boolean
*/
function hasPrev() {

}

/**
```

```
* @return Integer
*/
function prev() {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* $obj = BSTIterator($root);
* $ret_1 = $obj->hasNext();
* $ret_2 = $obj->next();
* $ret_3 = $obj->hasPrev();
* $ret_4 = $obj->prev();
*/
```

Dart Solution:

```
/**
* Definition for a binary tree node.
* class TreeNode {
* int val;
* TreeNode? left;
* TreeNode? right;
* TreeNode([this.val = 0, this.left, this.right]);
* }
*/
class BSTIterator {

BSTIterator(TreeNode? root) {

}

bool hasNext() {

}

int next() {

}
```

```
bool hasPrev() {

}

int prev() {

}
}

/**
* Your BSTIterator object will be instantiated and called as such:
* BSTIterator obj = BSTIterator(root);
* bool param1 = obj.hasNext();
* int param2 = obj.next();
* bool param3 = obj.hasPrev();
* int param4 = obj.prev();
*/
```

Scala Solution:

```
/**
* Definition for a binary tree node.
* class TreeNode(_value: Int = 0, _left: TreeNode = null, _right: TreeNode =
null) {
* var value: Int = _value
* var left: TreeNode = _left
* var right: TreeNode = _right
* }
*/
class BSTIterator(_root: TreeNode) {

def hasNext(): Boolean = {

}

def next(): Int = {

}

def hasPrev(): Boolean = {
```

```
}

def prev(): Int = {

}

}

/**
* Your BSTIterator object will be instantiated and called as such:
* val obj = new BSTIterator(root)
* val param_1 = obj.hasNext()
* val param_2 = obj.next()
* val param_3 = obj.hasPrev()
* val param_4 = obj.prev()
*/
```

Elixir Solution:

```
# Definition for a binary tree node.
#
# defmodule TreeNode do
# @type t :: %__MODULE__{
# val: integer,
# left: TreeNode.t() | nil,
# right: TreeNode.t() | nil
# }
# defstruct val: 0, left: nil, right: nil
# end

defmodule BSTIterator do
@spec init_(root :: TreeNode.t | nil) :: any
def init_(root) do

end

@spec has_next() :: boolean
def has_next() do

end
```

```
@spec next() :: integer
def next() do

end

@spec has_prev() :: boolean
def has_prev() do

end

@spec prev() :: integer
def prev() do

end
end

# Your functions will be called as such:
# BSTIterator.init_(root)
# param_1 = BSTIterator.has_next()
# param_2 = BSTIterator.next()
# param_3 = BSTIterator.has_prev()
# param_4 = BSTIterator.prev()

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

Erlang Solution:

```
%% Definition for a binary tree node.
%%
%% -record(tree_node, {val = 0 :: integer(),
%% left = null :: 'null' | #tree_node{},
%% right = null :: 'null' | #tree_node{}}).

-spec bst_iterator_init_(Root :: #tree_node{} | null) -> any().
bst_iterator_init_(Root) ->
.

-spec bst_iterator_has_next() -> boolean().
bst_iterator_has_next() ->
.
```

```
-spec bst_iterator_next() -> integer().
bst_iterator_next() ->
.

-spec bst_iterator_has_prev() -> boolean().
bst_iterator_has_prev() ->
.

-spec bst_iterator_prev() -> integer().
bst_iterator_prev() ->
.


%% Your functions will be called as such:
%% bst_iterator_init_(Root),
%% Param_1 = bst_iterator_has_next(),
%% Param_2 = bst_iterator_next(),
%% Param_3 = bst_iterator_has_prev(),
%% Param_4 = bst_iterator_prev(),

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

Racket Solution:

```
; Definition for a binary tree node.
#|

; val : integer?
; left : (or/c tree-node? #f)
; right : (or/c tree-node? #f)
(struct tree-node
(val left right) #:mutable #:transparent)

; constructor
(define (make-tree-node [val 0])
(tree-node val #f #f))

|#
```

```
(define bst-iterator%
(class object%
(super-new)

; root : (or/c tree-node? #f)
(init-field
root)

; has-next : -> boolean?
(define/public (has-next)
)
; next : -> exact-integer?
(define/public (next)
)
; has-prev : -> boolean?
(define/public (has-prev)
)
; prev : -> exact-integer?
(define/public (prev)
)))

;; Your bst-iterator% object will be instantiated and called as such:
;; (define obj (new bst-iterator% [root root]))
;; (define param_1 (send obj has-next))
;; (define param_2 (send obj next))
;; (define param_3 (send obj has-prev))
;; (define param_4 (send obj prev))
```