Custom Binary Search Tree in JavaScript
v1.0
Cpt S 489

(read all instructions carefully before writing any code; do all work individually!)

The zip that contains these instructions also includes an HTML and JavaScript code file. The HTML page is set up to reference the JavaScript code file and use it to run tests. Do not change anything in this file. The JavaScript file is where you must write the code.

Implement a class for a binary search tree such that you can create the tree with a line of code like:

```
var tree = new BST(compareFunction);
```

or

```
var tree = new BST();
```

and then access any of the members listed on the next page. The comparison function is optional and must default to the code listed in the table on the next page if null or undefined. Use this function to compare values when adding/removing/searching in your BST. For details about how this comparison function works, see the description of the comparison function described on the Array.prototype.sort page on MDN:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort

Each node in the tree contains the following properties:

- value
- left
- right

and if you do the challenge case, additional properties:

- next
- previous

If you wish you can add a **parent** member to nodes as well. This may make things easier for a future assignment but is not required for this assignment.

Match casing for each of these members. If you decide not to do the challenge cases, you may omit the corresponding members (**next** and **previous** from nodes, **m_first** and **m_last** from tree).

Members of the BST class are described on the next page.

	Name	Description
Basic	m_root	Reference to the root node of the tree. Null when the tree is empty.

	m_compare	Reference to a function that compares values that are to be inserted into nodes in the tree. Defaults to the following value if a null or undefined reference is passed to the constructor:
		`function(a, b)`
		`{`
		`  if (a < b) { return -1; }`
		`  else if (a > b) { return 1; }`
		`  return 0;`
		`};`

	add	Function that takes a single parameter for a value to add to the tree. Do not allow duplicates. Return true if the value is successfully added into the tree, false otherwise. Use the standard BST insertion rules. Do not balance the tree in any way.
	count	Function that is callable with no parameters that returns the number of values in the tree. Returns 0 if the tree is empty.
	getLevel	Function that takes a single parameter for a value to search for in the tree. If the value is found, the level index of the node containing the value is returned. This is a zero-based index, so the root is on level 0, its children on level 1, and so on. If the value is not in the tree, then return -1.
	getMax	Function that is callable with no parameters that returns the maximum value in the tree. Return undefined if the tree is empty.
	getMin	Function that is callable with no parameters that returns the minimum value in the tree. Return undefined if the tree is empty.
	has	Function that takes a single parameter for a value. Returns true if the value is in the tree, false if it is not.
	remove	Function that takes a single value to remove from the tree. Use the removal rules discussed in class (which are the standard removal rules). Opt for the max value in the left subtree when you need to do a swap for a removal of a node with 2 children. Return true if the value is found and removed, false otherwise.
	toString	Function that has a single, optional parameter for a delimiter string. If null or undefined, this delimiter defaults to a single space. Produces a string representation of the tree contents. The contents are in sorted order and separated by the delimiter string. The delimiter is space by default but can be replaced with any arbitrary string. **Important:** The delimiter is NOT included after the last element.
Challenge	m_first	(see description of challenge point)
	m_last	
	forEach	

Remember that names must match exactly including casing. There are 3 groups of tests on the web page and you must pass ALL of the tests in the first 2 groups to get 2 points out of 3. You may add additional utility functions to the tree if you wish.

Challenge Point: Tracking Insertion Order

For the 3rd point on this assignment, you must keep track of the insertion order of elements in the tree and support iteration in insertion order via a **forEach** function. We are working up to, on a later assignment, providing an replacement for the Set object in ES6. Our BST could actually emulate its functionality very well. The ES6 Set has a forEach function described here:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/forEach

Our forEach function is similar overall, with a few small differences. It has a first parameter that is a callback function and a second parameter that is a bool indicating whether to iterate through in insertion order. If the second parameter is false or omitted, then it defaults to iterating in sorted order (standard in-order traversal). An excerpt from the function declaration line follows.

```
forEach = function(callback, useInsertionOrder)
```

When you invoke the callback function in your forEach function implementation, give it the value of the node as the first parameter and the reference to the tree as the second parameter.

Example:
Suppose you insert 33, 55, and 22, in that order, into your BST. If the forEach function were called with the second parameter being false or omitted, the callback would be invoked in the order: 22, 33, 55. If forEach was instead called with the second parameter being true, the callback would be invoked in the order: 33, 55, 22.

To do this, you must "hybridize" the tree to include a doubly-linked-list that keeps track of insertion order. This is NOT a separate list stored as another member of the tree. Instead, each node in the tree gains a **next** and **previous** member, and each node in the tree is both part of a BST and a doubly-linked-list. The tree object gains **m_first** and **m_last** members, which reference the first node and last node inserted, respectively. The processing of adding or removing a node in the tree is extended to include the construction/alteration of the linked list.

All operations for adding or removing one item from this linked list must operate in constant time. This means you aren't increasing the time complexity of the tree operations.

Scoring:
All tests pass: 3/3
All tests from the first 2 groups pass, but 1 or more fails in the 3rd group: 2/3
One or more tests from the first 2 groups fail but still 50% or more of such tests pass: 1/3
Anything else: 0/3