Lab Sheet 6

"Treat your password like your toothbrush.
Don't let anybody else use it, and
get a new one every six months.
--Clifford Stoll

1. A *Binary Search Tree* (BST) is a binary tree in which each internal node `x` stores an element such that each element stored in the left subtree of `x` are less than that of `x` and the elements stored in the right subtree of `x` are greater than `x`. BSTs are a fundamental data structure used to construct more abstract data structures such as *sets*, *multisets*, *maps*, *associative arrays*, etc.

   Construct a class to represent a binary search tree for storing integer values. The name of the class may be `Tree_Int`. The class should be designed as follows:

   Data Members:

Name of data member	Description
`root`	The pointer to the root of the tree. Type should be `Node<int> *`. A `Node` object has a *data* part, and a *left* & *right subtree* pointers.
`tree_size`	Number of elements stored in tree. Type should be `size_t`.

   Constructor:

Constructor	Description
`Tree_Int()`	The default constructor. Set the root to NULL and the size of the tree to 0.

   Destructor:

Destructor	Description
`~Tree_Int()`	The destructor should deallocate the dynamically allocated nodes. This can be done by calling the `clear()` function. (See below).

   Member functions:

Function name	Description
`void insert(int x);`	To insert a new node to the tree.
`Node<int>* get_root() const;`	To get the root of the tree.
`size_t size() const;`	To get the number of elements in the tree.
`bool empty() const;`	To check whether empty.

`void inorder (Node<int> *node_ptr, ostream& out = cout) const ;`
 	
`void inorder(ostream& out = cout) const;`
	Traversing the tree using inorder traversal scheme. The first version traverses the tree from any particular node and the second version traverses from the root node.

`void preorder (Node<int> *node_ptr, ostream& out = cout) const ;`	Traversing the tree using preorder traversal scheme. The first version traverses the tree from any particular

`void preorder(ostream& out = cout) const;`	node and the second version traverses from the root node.

`void postorder`
`(Node<int> *node_ptr, ostream& out = cout)`
`const ;`
 	
`void postorder(ostream& out = cout) const;`
	Traversing the tree using postorder traversal scheme. The first version traverses the tree from any particular node and the second version traverses from the root node.

`void levelorder`
`(Node<int> *node_ptr, ostream& out = cout)`
`const ;`
 	
`void levelorder(ostream& out = cout) const;`
	Traversing the tree using levelorder traversal scheme. The first version traverses the tree from any particular node and the second version traverses from the root node.

`Node<int>* search(int key);`	To search for a particular key.
`bool delete_node(int value);` `bool delete_node(Node<int>* node);`	To delete a node with a specific value.
`Node<int>* get_parent(Node<int>* node);` `Node<int>* get_parent(int value);`	To get the parent of a particular node.

`Node<int>*`
`get_inorder_successor(Node<int> *node);`
 	
`Node<int>*`
`get_inorder_successor(int value);`
	To get the inorder successor given a pointer to a node in the tree or a value of a node in the tree. Returns NULL if there is not inorder successor for the given node.

`Node<int>*`
`get_inorder_predecessor(Node<int> *node);`
 	
`Node<int>*`
`get_inorder_predecessor(int value);`
	To get the inorder predecessor given a pointer to a node in the tree or a value of a node in the tree. Returns NULL if there is not inorder predecessor for the given node.

`bool is_left_child (Node<int>* node);` `bool is_left_child (int value);`	To check whether a node is a left child of its parent. For the root node, return NULL.
`bool is_right_child(Node<int>* node);` `bool is_right_child(int value);`	To check whether a node is a right child of its parent. For the root node, return NULL.
`bool is_root(Node<int>* node);` `bool is_root(int value);`	To check whether a given node is a root node.
`bool has_two_siblings(Node<int>* node);` `bool has_two_siblings(int value);`	To check whether a given node has two children.
`bool is_leaf(Node<int>* node);` `bool is_leaf(int value);`	To check whether a given node is a leaf node.
`bool has_only_one_sibling(Node<int>* node);` `bool has_only_one_sibling(int value);`	To check whether a given node has only a single child.
`void clear();` `void clear(Node<int>* node);`	To clear the entire tree or a particular subtree of the entire tree.

Which of the above functions would you prefer to maintain as *utility functions* (in the *private* section)?

Write a *driver* to check for the functionalities of the above class.