Assignment 17: Binary Search Trees

This assignment is an exercise in implementing the map ADT using a binary search tree. You will need to write a template structure to represent a tree node, and a template class to represent the tree.

1. Initial Setup

1. Log in to Unix.
2. Run the `setup` script for Assignment 17 by typing:

   ```
   setup 17
   ```

2. Files You Must Write

You will need to write one template structure and one template class for this assignment. A main program to test your class will be provided.

Since this is a C++ template, all of the code for both the structure and the class should be placed in a single file, `bstree.h`. This is the only file that you need to submit to the Grade-o-Matic and on Unix. It will include both structure / class definitions as well as the definitions for all of the functions. See the **Hints** section for an outline of how to structure this file and the order that things need to be coded.

The `node` structure

Data members

This template structure should have four data members: a member of the template parameter type used to store a key, a member of the template parameter type `T` used to store a value, a pointer to a `node<K, V>` that points to the node's left subtree, and a pointer to a `node<K, V>` that point's to the node's right subtree.

Member functions

- Constructor

  The structure should have a constructor that can be used to initialize the data members of the tree node.

The `bstree` class

Data members

This class should have two data members. The first should be a pointer to a `node<K, V>` that will point to the root node of the tree (or be `nullptr` if the tree is empty). The second should be a `size_t` variable that will be used to store the tree size, the number of elements or values currently stored in the binary search tree.

Member functions

The `bstree` class should implement the following member functions:

- ```
  template <class K, class V>
  bstree<K, V>::bstree()
  ```

  The class should have a default constructor that sets the root pointer data member of the tree to `nullptr` and the tree size to 0.

- ```
  template <class K, class V>
  bstree<K, V>::~bstree()
  ```

  The class should have a destructor. The destructor can simply call the `clear()` method.

- ```
  template <class K, class V>
  bstree<K, V>::bstree(const bstree<K, V>& x)
  ```

  The class should have a proper copy constructor. Making a copy of the tree nodes can be done by performing a modified preorder traversal as described in the course notes on binary search trees.

- ```
  template <class K, class V>
  bstree<K, V>& bstree<K, V>::operator=(const bstree<K, V>& x)
  ```

  The assignment operator should be properly overloaded.

- ```
  template <class K, class V>
  void bstree<K, V>::clear()
  ```

  This member function should set the tree back to the empty state, deleting all of the nodes in the tree and setting the size back to 0.

- ```
  template <class K, class V>
  size_t bstree<K, V>::size() const
  ```

  Returns the tree size.

- ```
  template <class K, class V>
  size_t bstree<K, V>::height() const
  ```

  Returns the tree height. In this assignment, we will use a slightly modified definition for the height of a tree. An empty tree will have height 0, a tree with only a root node will have height 1. The height of any other node in the tree is the height of its parent node + 1. The height of the tree can then be defined as the maximum height of any of its nodes.

- ```
  template <class K, class V>
  bool bstree<K, V>::empty() const
  ```

  Returns true if the tree size is 0; otherwise, returns false.

- ```
  template <class K, class V>
  const K& bstree<K, V>::min() const
  ```

  This member function should return the minimum key in the tree. You may assume that this function will not be called for an empty tree.

- ```
  template <class K, class V>
  const K& bstree<K, V>::max() const
  ```

  This member function should return the maximum key in the tree. You may assume that this function will not be called for an empty tree.

- ```
  template <class K, class V>
  bool bstree<K, V>::insert(const K& key, const V& value)
  ```

  This member function should attempt to insert a key and value into the binary search tree. If the key already exists in the tree, the function should return false. Otherwise, a new tree node containing the key and value should be inserted in the correct spot to
  ```

maintain the ordered property of the binary search tree, the size of the tree should be incremented, and the function should return true.

- `template <class K, class V>`
  `bool bstree<K, V>::remove(const K& key)`

  This member function should attempt to remove the specified key from the binary search tree. If the key is not in the tree, the function should return false. Otherwise, the node with a matching key should be removed, the size of the tree should be decremented, and the function should return true.

- `template <class K, class V>`
  `const node<K, V>* bstree<K, V>::find(const K& key) const`

  This member function should attempt to find the specified key in the binary search tree. If the key is not in the tree, the function should return `nullptr`. Otherwise, it should return the address of the node that contains the specified key.

- `template <class K, class V>`
  `void bstree<K, V>::preorder() const`

  This member function should perform a preorder traversal of the tree from left to right. As each node is visited, it should have its key and value printed (see **Output** below for the required format).

- `template <class K, class V>`
  `void bstree<K, V>::inorder() const`

  This member function should perform a inorder traversal of the tree from left to right. As each node is visited, it should have its key and value printed (see **Output** below for the required format).

- `template <class K, class V>`
  `void bstree<K, V>::postorder() const`

  This member function should perform a preorder traversal of the tree from left to right. As each node is visited, it should have its key and value printed (see **Output** below for the required format).

- `template <class K, class V>`
  `void bstree<K, V>::level_order() const`

  This member function should perform a level order traversal of the tree from left to right. As each node is visited, it should have its key and value printed (see **Output** below for the required format).

  This member function should be written non-recursively. Pseudocode for a recursive implementation of the function is available on the course website, but it's frankly not a good solution to the problem.

3. Files We Give You

The setup script on Unix will create the directory `Assign11` under your `csci241` directory. It will copy a makefile named `makefile` to the assignment directory.

You will also receive a driver program named `main.cpp` which contains a `main()` function that will test your `bstree` class.

4. Files You Must Submit

The only file you must submit to the Grade-o-Matic and on Unix is `bstree.h`.

5. Output

The correct output for this assignment is shown below:

```
*** Testing default constructor ***

*** Testing size(), height(), empty(), and find() for empty tree ***

size(): 0
height(): 0
tree is empty
56 not found

*** Testing traversals of empty tree ***

preorder:


inorder:
```

```
postorder:


level order:


*** Testing insertion of root node ***

*** Testing size(), height(), empty(), min(), max(), and find() for tree
with one node ***

size(): 1
height(): 1
tree is not empty
Minimum key: 56
Maximum key: 56

56 found, value: 7.50
39 not found

*** Testing traversals of tree with one node ***

preorder:

56: 7.50

inorder:

56: 7.50

postorder:

56: 7.50

level order:

56: 7.50

*** Testing insertion of additional nodes ***

*** Testing size(), height(), empty(), min(), max(), and find() ***

size(): 9
height(): 4
tree is not empty
Minimum key: 21
Maximum key: 86

56 found, value: 7.50
```

```
39 found, value: 8.99
65 not found

*** Testing traversals of tree ***

preorder:

56: 7.50
34: 2.27
21: 5.12
45: 3.91
39: 8.99
68: 6.49
62: 1.00
74: 7.03
86: 9.74

inorder:

21: 5.12
34: 2.27
39: 8.99
45: 3.91
56: 7.50
62: 1.00
68: 6.49
74: 7.03
86: 9.74

postorder:

21: 5.12
39: 8.99
45: 3.91
34: 2.27
62: 1.00
86: 9.74
74: 7.03
68: 6.49
56: 7.50

level order:

56: 7.50
34: 2.27
68: 6.49
21: 5.12
45: 3.91
62: 1.00
74: 7.03
39: 8.99
```

```
86: 9.74

*** Testing insertion of duplicate keys ***

*** Insert duplicate key 56 ***

Failure
size(): 9

*** Insert duplicate key 39 ***

Failure
size(): 9

*** Testing const correctness ***

size(): 9
height(): 4
tree is not empty
Minimum key: 21
Maximum key: 86

39 found, value: 8.99

preorder:

56: 7.50
34: 2.27
21: 5.12
45: 3.91
39: 8.99
68: 6.49
62: 1.00
74: 7.03
86: 9.74

inorder:

21: 5.12
34: 2.27
39: 8.99
45: 3.91
56: 7.50
62: 1.00
68: 6.49
74: 7.03
86: 9.74

postorder:

21: 5.12
```

```
39: 8.99
45: 3.91
34: 2.27
62: 1.00
86: 9.74
74: 7.03
68: 6.49
56: 7.50

level order:

56: 7.50
34: 2.27
68: 6.49
21: 5.12
45: 3.91
62: 1.00
74: 7.03
39: 8.99
86: 9.74

*** Testing copy constructor ***

size(): 0
height(): 0

preorder:


inorder:


size(): 9
height(): 4

preorder:

56: 7.50
34: 2.27
21: 5.12
45: 3.91
39: 8.99
68: 6.49
62: 1.00
74: 7.03
86: 9.74

inorder:

21: 5.12
34: 2.27
```

```
39: 8.99
45: 3.91
56: 7.50
62: 1.00
68: 6.49
74: 7.03
86: 9.74

*** Testing clear() ***

size(): 0
height(): 0

*** Testing for shallow copy ***

size(): 9
height(): 4

*** Testing copy assignment operator ***

size(): 0
height(): 0

preorder:


inorder:


size(): 9
height(): 4

preorder:

56: 7.50
34: 2.27
21: 5.12
45: 3.91
39: 8.99
68: 6.49
62: 1.00
74: 7.03
86: 9.74

inorder:

21: 5.12
34: 2.27
39: 8.99
45: 3.91
56: 7.50
```

```
62: 1.00
68: 6.49
74: 7.03
86: 9.74

*** Testing assignment to self ***

size(): 9
height(): 4

*** Testing chained assignment ***

size(): 3
height(): 3

preorder:

8: 1.00
10: 2.00
27: 3.00

inorder:

8: 1.00
10: 2.00
27: 3.00

size(): 3
height(): 3

preorder:

8: 1.00
10: 2.00
27: 3.00

inorder:

8: 1.00
10: 2.00
27: 3.00

*** Testing removal of non-existent node ***

Failure
size(): 13

*** Testing removal of node with no children ***

Success
size(): 12
```

```
preorder:

56: 7.50
34: 2.27
45: 3.91
39: 8.99
68: 6.49
62: 1.00
65: 1.00
64: 7.00
74: 7.03
72: 3.00
73: 5.00
86: 9.74

inorder:

34: 2.27
39: 8.99
45: 3.91
56: 7.50
62: 1.00
64: 7.00
65: 1.00
68: 6.49
72: 3.00
73: 5.00
74: 7.03
86: 9.74

*** Testing removal of node with only right child ***

Success
size(): 11

preorder:

56: 7.50
34: 2.27
45: 3.91
39: 8.99
68: 6.49
62: 1.00
65: 1.00
64: 7.00
74: 7.03
73: 5.00
86: 9.74

inorder:
```

```
34: 2.27
39: 8.99
45: 3.91
56: 7.50
62: 1.00
64: 7.00
65: 1.00
68: 6.49
73: 5.00
74: 7.03
86: 9.74

*** Testing removal of node with only left child ***

Success
size(): 10

preorder:

56: 7.50
34: 2.27
39: 8.99
68: 6.49
62: 1.00
65: 1.00
64: 7.00
74: 7.03
73: 5.00
86: 9.74

inorder:

34: 2.27
39: 8.99
56: 7.50
62: 1.00
64: 7.00
65: 1.00
68: 6.49
73: 5.00
74: 7.03
86: 9.74

*** Testing removal of node with two children ***

Success
size(): 9

preorder:
```

```
56: 7.50
34: 2.27
39: 8.99
65: 1.00
62: 1.00
64: 7.00
74: 7.03
73: 5.00
86: 9.74

inorder:

34: 2.27
39: 8.99
56: 7.50
62: 1.00
64: 7.00
65: 1.00
73: 5.00
74: 7.03
86: 9.74

*** Testing removal of root node with two children ***

Success
size(): 8

preorder:

39: 8.99
34: 2.27
65: 1.00
62: 1.00
64: 7.00
74: 7.03
73: 5.00
86: 9.74

inorder:

34: 2.27
39: 8.99
62: 1.00
64: 7.00
65: 1.00
73: 5.00
74: 7.03
86: 9.74
```

6. Hints

- The driver program is designed to make this assignment easy to develop incrementally. Simply comment out all of the lines of the driver program that call functions that you haven't written yet. You should be able to write, test, and debug one function at a time.

- This assignment sheet describes the "public interface" for the binary search tree class. You are welcome (and encouraged) to write additional private member functions for your class as needed.

- Pseudocode for many of the tree member functions can be found in the notes on Binary Tree Traversals and Binary Search Trees on the course web site.