Binary Search Trees

In this coursework you will be making a Binary Search Tree template class. As with previous weeks, the coursework has two deadlines: one for the main part, and one for the advanced part.

Note that in answering these questions, you should not add any `#include` or `using` statements: you must implement the functionality yourself, without using any additional data structures from the standard library. I have added `#include <memory>`for `std::unique_ptr`, `#include <utility>` for `std::pair`, and `#include <iostream>`. If you have a convincing need for adding a different `#include` please post in the forum on KEATS.

Binary Search Trees, Main Part (7 marks)

a) Making a tree node

In the file `treenode.h` implement a template class `TreeNode` that represents a node in a binary search tree. It should have four `public` member variables:

- The `data` stored in that node. The type of this should be a template argument.
- A unique_ptr for the left child of the node;
- A unique_ptr for the right child of the node;
- A pointer to the parent of the node (NB not a unique_ptr)

Make a constructor that takes an item of data, stores it in the node, and sets the parent pointer to nullptr.

Make a function setLeftChild(TreeNode* child) that:

- stores child in the left unique_ptr by calling reset() on it
- sets the parent pointer of the child to point to this

Write an analogous function setRightChild for setting the right child of the node.

Make a function write that takes an ostream reference, and prints to it:

- The result of calling write on the left child (if there is one)
- A space character
- The data from the node
- A space character
- The result of calling write on the right child (if there is one)

You should then be able to write, e.g:

```
someNode->write(cout);
```

...to be able to print the subtree starting at some node to screen. (NB write should be marked as const.)

To test your code, compile and run TestTreeNode.cpp. A Makefile has been provided, run:

```
make TestTreeNode
```

...at the command line. This makes four tree nodes, linked to each other, then prints out the tree.

b) Making a tree

In the file `tree.h` implement a template class BinarySearchTree. This should use the `TreeNode` class you have written so far.

As a private member variable you should have a unique_ptr, `root`, containing a pointer to the root TreeNode.

write

Write a function `write` that takes an `ostream` reference, and calls `write` on the root of the tree. (NB write should be marked as const.)

insert

Make a function `insert` that takes an item of data, and inserts it into the tree:

- If the data is not already in the tree, it should make a node and add this in the correct place
- If something equal to the data is already in the tree, it shouldn't make any new nodes, and shouldn't change any of the existing nodes.

In both cases, it should return a `TreeNode*`, pointing to the node containing the data.

Note, in your implementation, you should only compare the data inside nodes by using the < operator. Do not use > or !=or == or any other operator. For each node, if it has a left child, then left->data < data, and if it has a right child, then data < right->data.

As an example, if the binary search tree is:

```
...4
./...\
2.....7
```

...and 3 is inserted, the tree should become:

```
...4
./...\
2.....7
.\
..3
```

..because, starting at the root:

- 3 < 4, so we need to go to the left
- 2 < 3, so we need to go to the right
- We've got to the bottom of the tree, so 3 is added as the right child of 2

A pointer to the node containing 3 would then be returned. (Not a unique_ptr, a normal pointer.)

If 7 is inserted into the tree above, the function should stop when it gets to the node containing 7, and return a pointer to that node.

find

Write a function `find` that takes an item of data and traverses the Binary Search Tree to see if the data is in the tree.

If it is, it should return a `TreeNode*` pointing to the node containing the data.

If it is not in the tree, it should return `nullptr`

To test your code, compile and run TestTree.cpp. A Makefile has been provided, run:

```
make TestTree
```

c) TreeMap

KeyValuePair

In the file `treemap.h`, the incomplete class `KeyValuePair` defines a class that holds a key--value pair, that will be used to make a map.

Complete the class by implementing:

- a Constructor that takes a Key and a Value and stores them in the respective member variables. *(NB Use the initialisation syntax for this)*
- a Constructor that takes just a Key, and stores this in the relevant member variable *(NB again, use the initialisation syntax)*
- an `operator<` function that compares it to another `KeyValuePair` object, by comparing just the keys (using the < operator). Remember to use const here correctly.

TreeMap

In the file treemap.h, the incomplete class TreeMap defines a class that holds a tree of key--value pairs.

Provided is a function insert that takes a Key and a Value, and inserts these into tree as a KeyValuePair. It then returns a pointer to the *data* inside the tree node returned by tree->insert.

Implement a function find that takes a Key, makes a KeyValuePair from it, and calls find on the tree to see if a match can be found. If it can be found, return a pointer to the *data* inside the tree node found by find (a KeyValuePair<Key,Value>*). If not, return nullptr.

To test your code, compile and run TestTreeMap.cpp. A Makefile has been provided, run:

```
make TestTreeMap
```

Binary Search Trees, Advanced Part [3 marks]

An iterator

In treenode.h implement a template class TreeNodeIterator that is an iterator over a binary search tree. As with the ListNodeIterator from the last practical, it should have:

- A single member variable pointing to a TreeNode *and no other member variables*

- A constructor that sets this to point to a given TreeNode
- An operator* that dereferences this pointer, and returns it (by reference)
- operator== and operator!= that compare it to other iterators, by checking if they point to the same (or a different) node
- An increment operator, operator++ that moves the iterator to point to the next node in the list.

For the tree:

```
...4
./...\
2.....7
.\
..3
```

- Incrementing an iterator pointing to 2 should make it point to 3;
- Incrementing an iterator pointing to 3 should make it point to 4;
- Incrementing an iterator pointing to 4 should make it point to 7

In other words, iterator steps through the tree in ascending order.

Extend your BinarySearchTree class with begin() and end() functions that return iterators to the left-most node in the tree (in the tree above -- 2), and nullptr, respectively.

maxDepth

In TreeNode implement a function maxDepth that returns the maximum depth of the subtree rooted at that node. If the TreeNode has no children, it has depth 1.

Otherwise, its depth is 1 + the maximum of either the depth of its left child, or the depth of its right child.

In `BinarySearchTree` implement a function maxDepth that returns the maxDepth of the root (or 0 for an empty tree).

AVL trees

In the worst case, when using a Binary Search Tree, the data is adding in ascending order, giving the following tree:

```
A
.\
..B
...\
....C
```

That is, the depth of the tree is the same as the number of elements. What we ideally want is a balanced tree, where the depth of the tree is *log(N)* in the number of elements, N.

AVL trees rebalance the tree every time a node is inserted. This is done by computing the *balance factor* of that node. It is computed as:

```
balanceFactor(node) = maxDepth(left node) - maxDepth(right node)
```

If this balance factor is ever 2, or -2, the tree beneath that node needs to be rebalanced: it is much deeper on one side than the other. For the following tree:

```
A
.\
..B
...\
....C
```

...the balance factors are:

```
-2
.\
..-1
...\
....0
```

...because looking at the root, the depth of the right subtree is 2; but the depth of the left subtree is 0.

To become rebalanced, an AVL tree performs one of four operations. Perform these where needed in your implementation of `insert` in the BinarySearchTree class. After inserting a node you will need to look at its parent, and its parent's parent; compute the balance factor of its parent's parent; and if the tree is then unbalanced, perform the appropriate operation.

(A full AVL tree implementation does a bit more than this, but implementing the cases described here is sufficient for this assignment.)

Left rotation

If a node becomes unbalanced, when a node is inserted into the right subtree of its right child, then we perform a left rotation. This is best shown with an example.

Suppose we had the subtree:

```
A
.\
..B
```

...and added 'C', we would get:

A
.\
..B
...\
....C

C is what we have just inserted; *B* is its parent; *A* is its parent's parent.

A now has a balance factor of -2, so we left rotate: we reorder the nodes so that B is the root of this subtree instead of A:

..B
./.\
A...C

Each of these now has a balance factor of 0, so it is balanced again.

Note if A had a parent, B is attached to this, replacing A.

Right rotation

If a node becomes unbalanced when a node is inserted into the left subtree of its left child, then we perform a right rotation. Suppose we had the tree:

....C
.../
..B

...and added 'A', we would get:

....C
.../
..B
./
A

C is now unbalanced: its balance factor is 2, because its left child has depth 2, but its right child is empty (depth 0). Thus, we right rotate: we reorder the nodes so that B is the root of this subtree instead of C:

```
..B
./.\
A...C
```

Note if C had a parent, B is attached to this, replacing C.

Left-Right rotation

If a node becomes unbalanced when a node is inserted into the right subtree of its left child, then we perform a left-right rotation. If we had the tree:

```
....C
.../
..A
```

...and added B, we would get:

```
....C
.../
..A
..\
...B
```

C is now unbalanced. This scenario is fixed by performing a left--right rotation. First, we perform a left rotation on the subtree rooted at A, making B the root of this subtree:

```
....C
.../
..B
./
A
```

Then, we perform a right rotation on C, making B the root of this subtree:

..B
./.\
A...C

Note if C had a parent, B is attached to this, replacing C.

Right-left rotation

One scenario left: a node becomes unbalanced when a node is inserted into the left subtree of its right child, then we perform a right-left rotation. If we had the tree:

....A
.....\
......C

... and added B, we would get:

....A
.....\
......C
...../
....B

A is now unbalanced. A right-left rotation fixes this in two stages. First, we perform a right rotation on the subtree rooted at C:

....A
.....\
......B
.......\
........C

Then, we perform a left rotation on A, making B the root of this subtree:

..B
./.\
A...C

Note if A had a parent, B is attached to this, replacing A.

To test your code, compile and run TestTreeD.cpp. A Makefile has been provided, run:

```
make TestTreeD
```