Exercises for 17 Jun 2020

1. Discuss the merits of using `std::shared_ptr` vs `std::unique_ptr` to manage the ownership of child nodes in a tree.

   Here are functions of unique_ptr and shared_ptr.

   1. unique_ptr: allows only one owner of the underlying pointer
   2. shared_ptr allows multiple owners of the same pointer (Reference count is maintained)

   Considering the definition of the tree. The pointer points to the child is better to use unique_ptr. Since the child only has one parent node.

2. Implement a tree ADT (*abstract data type*) in which each node can have an arbitrary number of children. Implement the following methods to populate your tree. Write test code to build a tree with some pre-defined structure and use a debugger to verify that the tree has the structure you intended.

```
/**
 * Set the value of the tree's root node.
 */
Tree& setRoot(T value);

/**
 * Add a leaf node to the top level of this tree.
 */
Tree& addChild(T value);

/**
 * Add a subtree to the top level of this tree, using move
 * semantics to "steal" the subtree's nodes.
 */
Tree& addSubtree(Tree<T>&&);
```

Exercises for 19 Jun 2020

Consider the alternate child-and-sibling structure described here.

1. Does this necessitate a change to your smart-pointer strategy (using `std::shared_ptr` or `std::unique_ptr`)? Why or why not?

   No. I can still use unique_ptr. because every sibling only points to one child.

2. Re-implement your tree data structure to use the child-and-sibling node structure.

   All changes are in Node class. the addChild method is changed. a new method named addSibling is added. Two field: child_ and sibling_ are added.

Exercises for 25 Jun 2020

1. Implement pre- and post-order traversal in your tree data structure, passing an instantiation of the following function template to your traversal method (i.e., your test code should call something like `myTree.visitPreorder(visit<double>)`):

```cpp
template<typename T>
void visit(const T &value)
{
    static size_t i = 0;
    std::cout << "Node " << i++ << ": " << value << "\n";
}
```

pseudocode:

```
def visit_preorder(node, fn):
    if node is None:
        return

    fn(node)
    for child in children:
        visit_preorder(child, fn)

def visit_postorder(node, fn):
    if node is None:
        return

    for child in children:
        visit_preorder(child, fn)
    fn(node)
```

2. [optional] Implement level-order traversal for your tree data structure. You may find the queue-based algorithm described on Wikipedia's "Tree Traversal" page to be helpful.

   skip