Iterator Pattern

Author: Jimmy Tran, Brian Crites

You must work in a group of two for this lab

The iterator pattern allows for traversal of various types of containers without the client knowing the underlying structure of the container. Therefore, the iterator's job is to simplify traversal and to remove the technical nuances of accessing a container allowing the developer to code at a higher level of abstraction. To make our iterator as reusable as possible we have to implement a core interface that all of our iterators will inherit.

In this lab, we will create a pre-order traversal iterator as well as the different collection iterators necessary to fully iterate our composite system. Our Iterator class will be used for both collection iterators and traversal iterators, so we want to keep it fairly basic

```
class Iterator {
    protected:
        Base* self_ptr;
        Base* current_ptr;

    public:
        Iterator (Base* ptr) { this->self_ptr = ptr; }

        /*Sets up the iterator to start at the beginning of traversal*/
        virtual void first() = 0;

        /*Move onto the next element*/
        virtual void next() = 0;

        /*Returns if you have finished iterating through all elements*/
        virtual bool is_done() = 0;

        /*Return the element the iterator is currently at*/
        virtual Base* current() = 0;
};
```

You will then need to write the following collection iterators

```
class OperatorIterator : public Iterator {
      public:
            OperatorIterator(Base* ptr);
```

```
            void first();
            void next();
            bool is_done();
            Base* current();
};
```

The OperatorIterator will be used to iterate over composite nodes with two children. This means it's first will initialize to the left child, and its next will cycle from left child (which is where it is set to start), to right child, then to NULL.

```
class UnaryIterator : public Iterator {
      public:
            UnaryIterator(Base* ptr);

            void first();
            void next();
            bool is_done();
            Base* current();
};
```

The UnaryIterator will be used to iterate over composite nodes with one child (only Sqr in our case). This means it's first will initialize to the only child (which has been redeclared in the composite class as the left child, with Unary having no right child to make for an easier interface), and next will cycle from child (which is where it is set to start) to NULL.

```
class NullIterator : public Iterator {
      public:
            NullIterator(Base* ptr);

            void first();
            void next();
            bool is_done();
            Base* current();
};
```

The NullIterator is used to iterate over leaf nodes. Since leaf nodes have no children, the NullIterator's is_done() will always return true and it's current() will always return NULL. It's first() and next() functions don't need to do anything.

You will also be required to implement a traversal iterator, in this case the pre-order traversal iterator (whose code structure can be referenced from the slides)

```
class PreOrderIterator : public Iterator {
	protected:
		stack<Iterator*> iterators;

	public:
		PreOrderIterator(Base* ptr);

		void first();
		void next();
		bool is_done();
		Base* current();
};
```

The PreOrderIterator has an additional stack data member, which we will use to keep tack of the collection iterators that we need to traverse. The rest of the functions will be written as follows

```
void PreOrderIterator::first() {
	//Empty the stack (just in case we had something leftover from
	//another run)

	//Create an iterator for the Base* that we built the iterator for

	//Initialize that iterator and push it onto the stack
}

void PreOrderIterator::next() {
	//Create an iterator for the item on the top of the stack

	//Initialize the iterator and push it onto the stack

	//As long as the top iterator on the stack is done, pop it off the
	//stack and then advance whatever iterator is now on top of the stack
}

bool PreOrderIterator::is_done() {
	//Return true if there are no more elements on the stack to iterate
}

Base* PreOrderIterator::current() {
	//Return the current for the top iterator in the stack
}
```

You can test your code with the provided test harness, however you should write additional tests to fully test your code for edge cases. Note that an additional Root node type has been added to the composite class. This is because the method above ignores the node used to initialize the traversal when iterating, so you can use this dummy node to have it iterate through the entire rest of the tree.

Once you have completed your PreOrderIterator, demo your traversal over several trees to your TA.