The Composite State Visitor Pattern

The Composite State Binary Tree with the Visitor Pattern

BiTreeCSV

The solution

The host system provides

- a minimal complete set of operations that any plugin developer can use to build the desired plugin
- an abstract visitor class with methods for the plugin developer to implement
- a public method named `accept()` for the end user to execute

The plugin developer

- writes a plugin by declaring a visitor to be a subclass of the abstract visitor class
- implements the visitor methods that the abstract visitor class provides

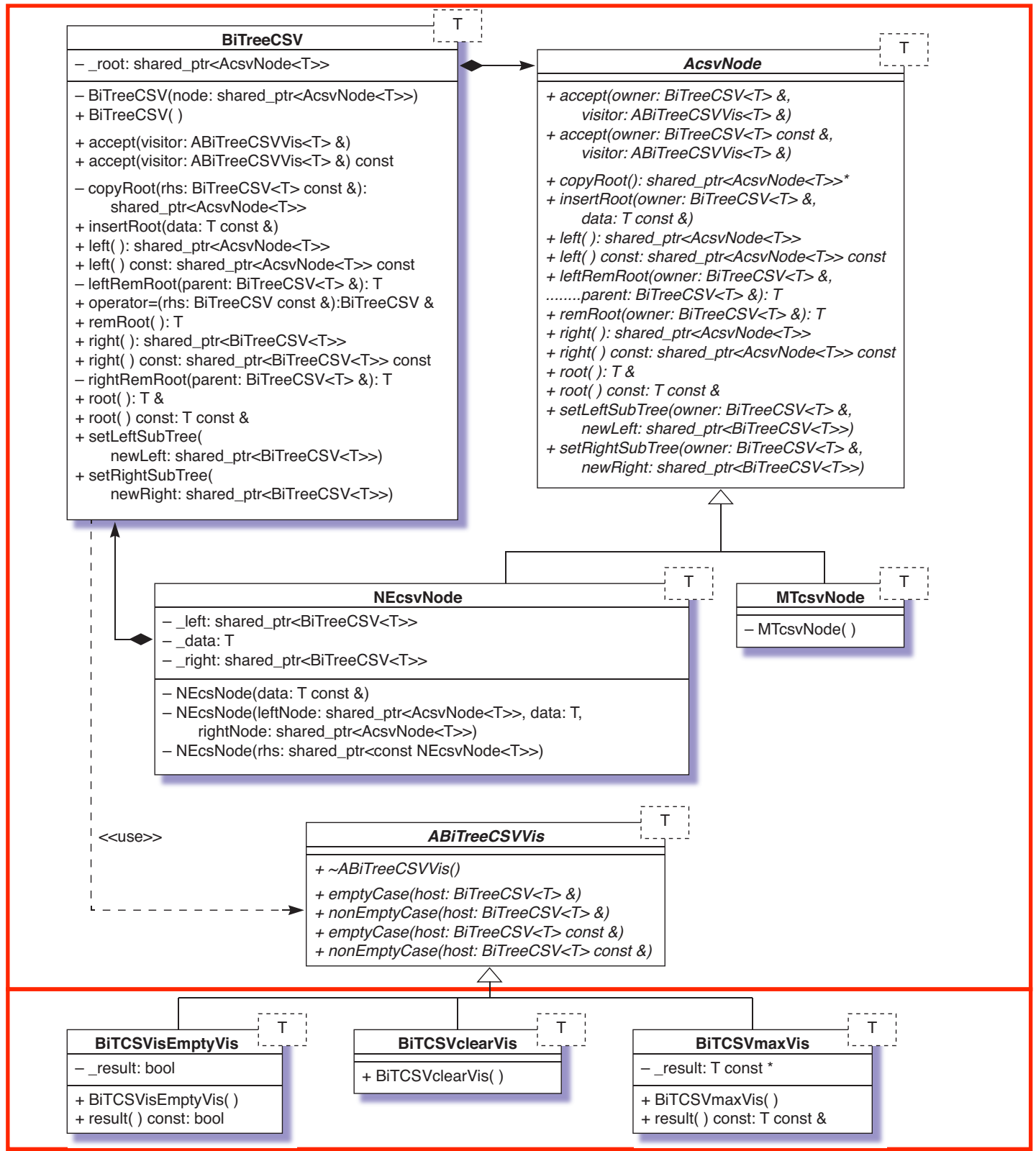
The plugin user

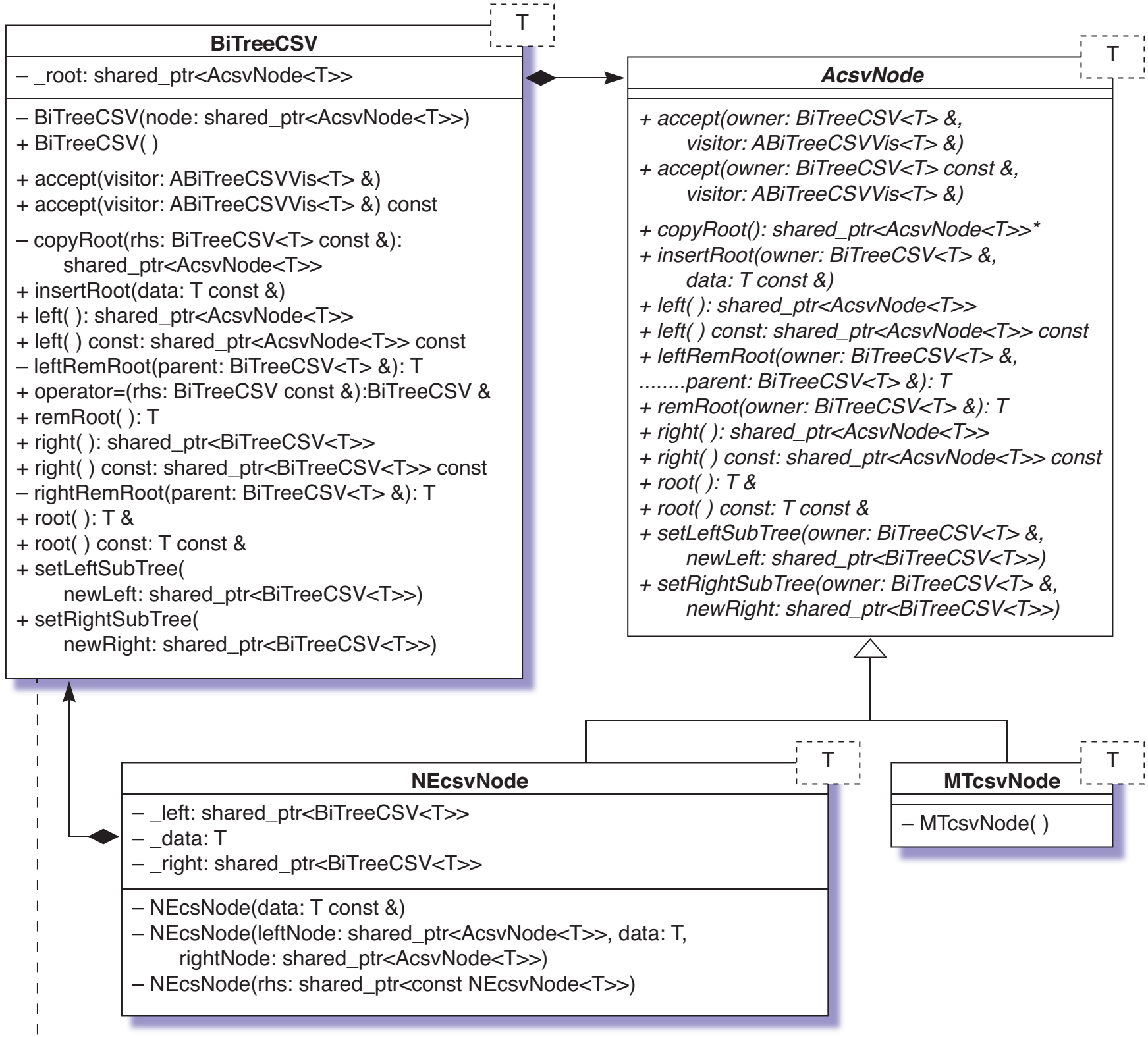
- instantiates a specific visitor object
- calls `accept()`, passing the visitor object as a parameter

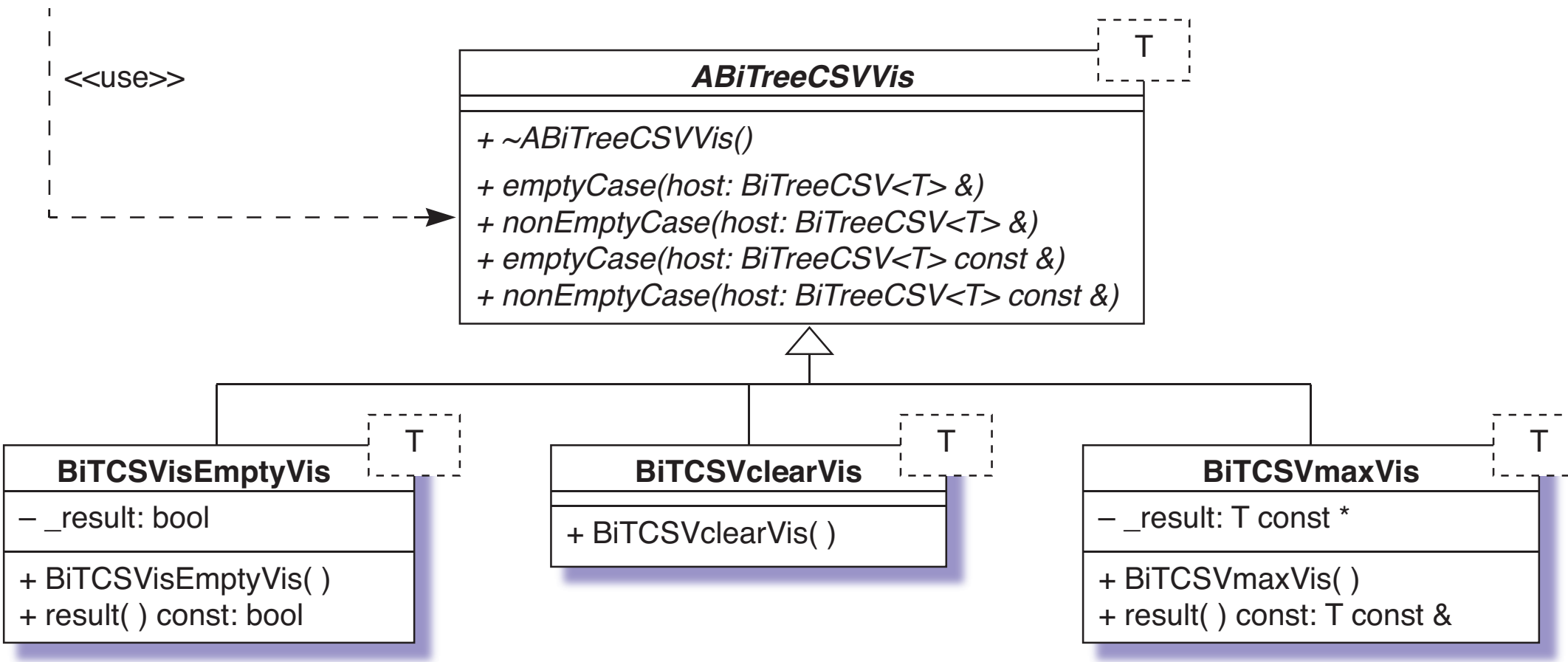
The host executes `accept()`, which, in turn, executes the methods implemented by the plugin developer.

Host system

Plugins




T
BiTreeCSV
– _root: shared_ptr<AcsvNode<T>>
– BiTreeCSV(node: shared_ptr<AcsvNode<T>>)
+ BiTreeCSV( )
+ accept(visitor: ABiTreeCSVVis<T> &)
+ accept(visitor: ABiTreeCSVVis<T> &) const
– copyRoot(rhs: BiTreeCSV<T> const &):
shared_ptr<AcsvNode<T>>
+ insertRoot(data: T const &)
+ left( ): shared_ptr<AcsvNode<T>>
+ left( ) const: shared_ptr<AcsvNode<T>> const
– leftRemRoot(parent: BiTreeCSV<T> &): T
+ operator=(rhs: BiTreeCSV const &):BiTreeCSV &
+ remRoot( ): T
+ right( ): shared_ptr<BiTreeCSV<T>>
+ right( ) const: shared_ptr<BiTreeCSV<T>> const
– rightRemRoot(parent: BiTreeCSV<T> &): T
+ root( ): T &
+ root( ) const: T const &
+ setLeftSubTree(
newLeft: shared_ptr<BiTreeCSV<T>>)
+ setRightSubTree(
newRight: shared_ptr<BiTreeCSV<T>>)
T
AcsvNode
+ accept(owner: BiTreeCSV<T> &,
visitor: ABiTreeCSVVis<T> &)
+ accept(owner: BiTreeCSV<T> const &,
visitor: ABiTreeCSVVis<T> &)
+ copyRoot(): shared_ptr<AcsvNode<T>>*
+ insertRoot(owner: BiTreeCSV<T> &,
data: T const &)
+ left( ): shared_ptr<AcsvNode<T>>
+ left( ) const: shared_ptr<AcsvNode<T>> const
+ leftRemRoot(owner: BiTreeCSV<T> &,
.....parent: BiTreeCSV<T> &): T
+ remRoot(owner: BiTreeCSV<T> &): T
+ right( ): shared_ptr<AcsvNode<T>>
+ right( ) const: shared_ptr<AcsvNode<T>> const
+ root( ): T &
+ root( ) const: T const &
+ setLeftSubTree(owner: BiTreeCSV<T> &,
newLeft: shared_ptr<BiTreeCSV<T>>)
+ setRightSubTree(owner: BiTreeCSV<T> &,
newRight: shared_ptr<BiTreeCSV<T>>)
T
NEcsvNode
– _left: shared_ptr<BiTreeCSV<T>>
– _data: T
– _right: shared_ptr<BiTreeCSV<T>>
– NEcsNode(data: T const &)
– NEcsNode(leftNode: shared_ptr<AcsvNode<T>>, data: T,
rightNode: shared_ptr<AcsvNode<T>>)
– NEcsNode(rhs: shared_ptr<const NEcsvNode<T>>)
T
MTcsvNode
– MTcsvNode( )


<<use>>
ABiTreeCSVVis
T
+ ~ABiTreeCSVVis()
+ emptyCase(host: BiTreeCSV<T> &)
+ nonEmptyCase(host: BiTreeCSV<T> &)
+ emptyCase(host: BiTreeCSV<T> const &)
+ nonEmptyCase(host: BiTreeCSV<T> const &)
BiTCSVisEmptyVis
T
– _result: bool
+ BiTCSVisEmptyVis( )
+ result( ) const: bool
BiTCSVclearVis
T
+ BiTCSVclearVis( )
BiTCSVmaxVis
T
– _result: T const *
+ BiTCSVmaxVis( )
+ result( ) const: T const &

Primitive methods

The primitive methods for BiTreeCSV are implemented similarly to the way they are for BiTreeCS.

```
// ========= accept =========
template<class T>
void BiTreeCSV<T>::accept(ABiTreeCSVVis<T> &visitor) {
    _root->accept(*this, visitor);
}

template<class T>
void MTcsvNode<T>::accept(BiTreeCSV<T> &owner, ABiTreeCSVVis<T> &visitor) {
    visitor.emptyCase(owner);
}

template<class T>
void NEcsvNode<T>::accept(BiTreeCSV<T> &owner, ABiTreeCSVVis<T> &visitor) {
    visitor.nonEmptyCase(owner);
}
```

```
// ========= accept =========
template<class T>
void BiTreeCSV<T>::accept(ABiTreeCSVVis<T> &visitor) {
    _root->accept(*this, visitor);
}

template<class T>
void MTcsvNode<T>::accept(BiTreeCSV<T> &owner, ABiTreeCSVVis<T> &visitor) {
    visitor.emptyCase(owner);
}

template<class T>
void NEcsvNode<T>::accept(BiTreeCSV<T> &owner, ABiTreeCSVVis<T> &visitor) {
    visitor.nonEmptyCase(owner);
}
```

The minimal complete set of methods

- To access the parts of a binary tree
  - `root()`
  - `left()`
  - `right()`
- To build a binary tree
  - `insertRoot()`
  - `setLeftSubtree()`
  - `setRightSubtree()`
  - `operator=()`
- To trim a binary tree
  - `remRoot()`

Implementing `remRoot()` in the host

There are two private helper methods

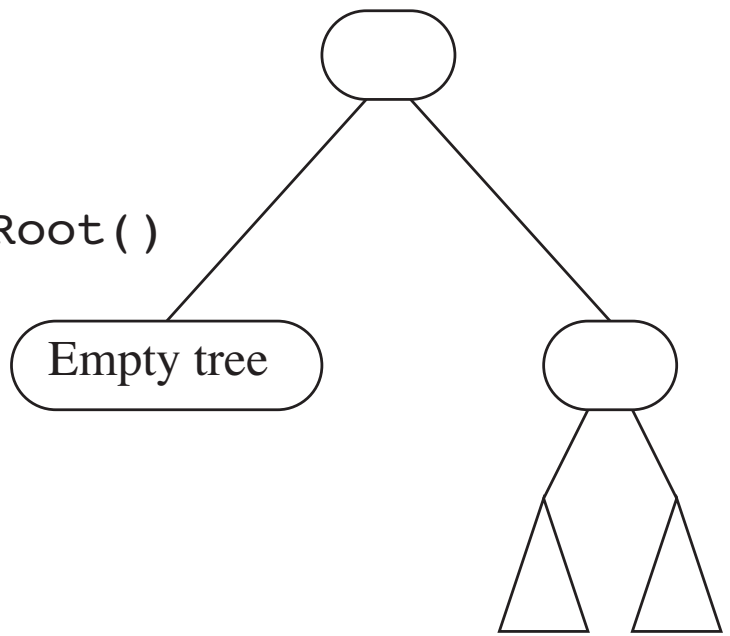
- `remRoot()` calls `leftRemRoot()`
- `leftRemRoot()` calls `rightRemRoot()`

```
// ========= remRoot =========
// Pre: This tree is not empty.
// Pre: The root of this tree has at least one empty child.
// Post: The root node is removed from this tree and its
// element is returned.

template<class T>
T BiTreeCSV<T>::remRoot() {
    return _root->remRoot(*this);
}

template<class T>
T MTcsvNode<T>::remRoot(BiTreeCSV<T> &owner) {
    cerr << "remRoot precondition violated: "
         << "Cannot remove root from an empty tree." << endl;
    throw -1;
}

template<class T>
T NEcsvNode<T>::remRoot(BiTreeCSV<T> &owner) {
    // owner is parent tree of _left.
    return _left->leftRemRoot(owner);
}
```

```
// ========= leftRemRoot =========
// Pre: Parent has at least one empty subtree.
// Post: The root of parent is removed and returned.
// Helper for remRoot.

template<class T>
T BiTreeCSV<T>::leftRemRoot(BiTreeCSV<T> &parent) {
    return _root->leftRemRoot(*this, parent);
}

template<class T>
T MTcsvNode<T>::leftRemRoot(BiTreeCSV<T> &owner, BiTreeCSV<T> &parent) {
    cerr << "MTcsvNode<T>::leftRemRoot: "
         << "Exercise for the student." << endl;
    throw -1;
}

template<class T>
T NEcsvNode<T>::leftRemRoot(BiTreeCSV<T> &owner, BiTreeCSV<T> &parent) {
    return parent.right()->rightRemRoot(parent);
}
```

```
// ========= rightRemRoot =========
// Pre: The left subtree of parent is not empty.
// Pre: The right subtree of parent is empty.
// Post: The root of parent is removed and returned.
// Helper for remRoot.

template<class T>
T BiTreeCSV<T>::rightRemRoot(BiTreeCSV<T> &parent) {
    return _root->rightRemRoot(*this, parent);
}

template<class T>
T MTcsvNode<T>::rightRemRoot(BiTreeCSV<T> &owner, BiTreeCSV<T> &parent) {
    cerr << "MTcsvNode<T>::rightRemRoot: "
         << "Exercise for the student." << endl;
    throw -1;
}

template<class T>
T NEcsvNode<T>::rightRemRoot(BiTreeCSV<T> &owner, BiTreeCSV<T> &parent) {
    cerr << "MTcsvNode<T>::rightRemRoot: "
         << "Exercise for the student." << endl;
    throw -1;
}
```

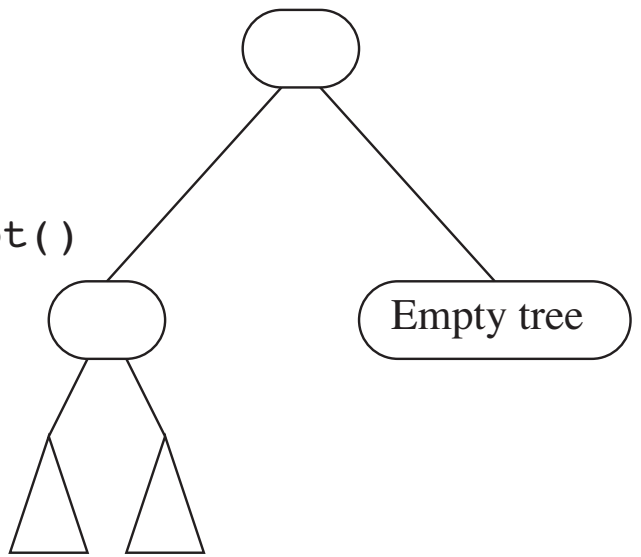
1. Tree `remRoot()` calls NENode `remRoot()`
2. NENode `remRoot()` calls left tree `leftRemRoot()`



3. Tree `leftRemRoot()` calls MTNode `leftRemRoot()`
4. MTNode `leftRemRoot()` removes the root

(a) Call sequence of `remRoot()` for an empty left subtree and a nonempty right subtree.

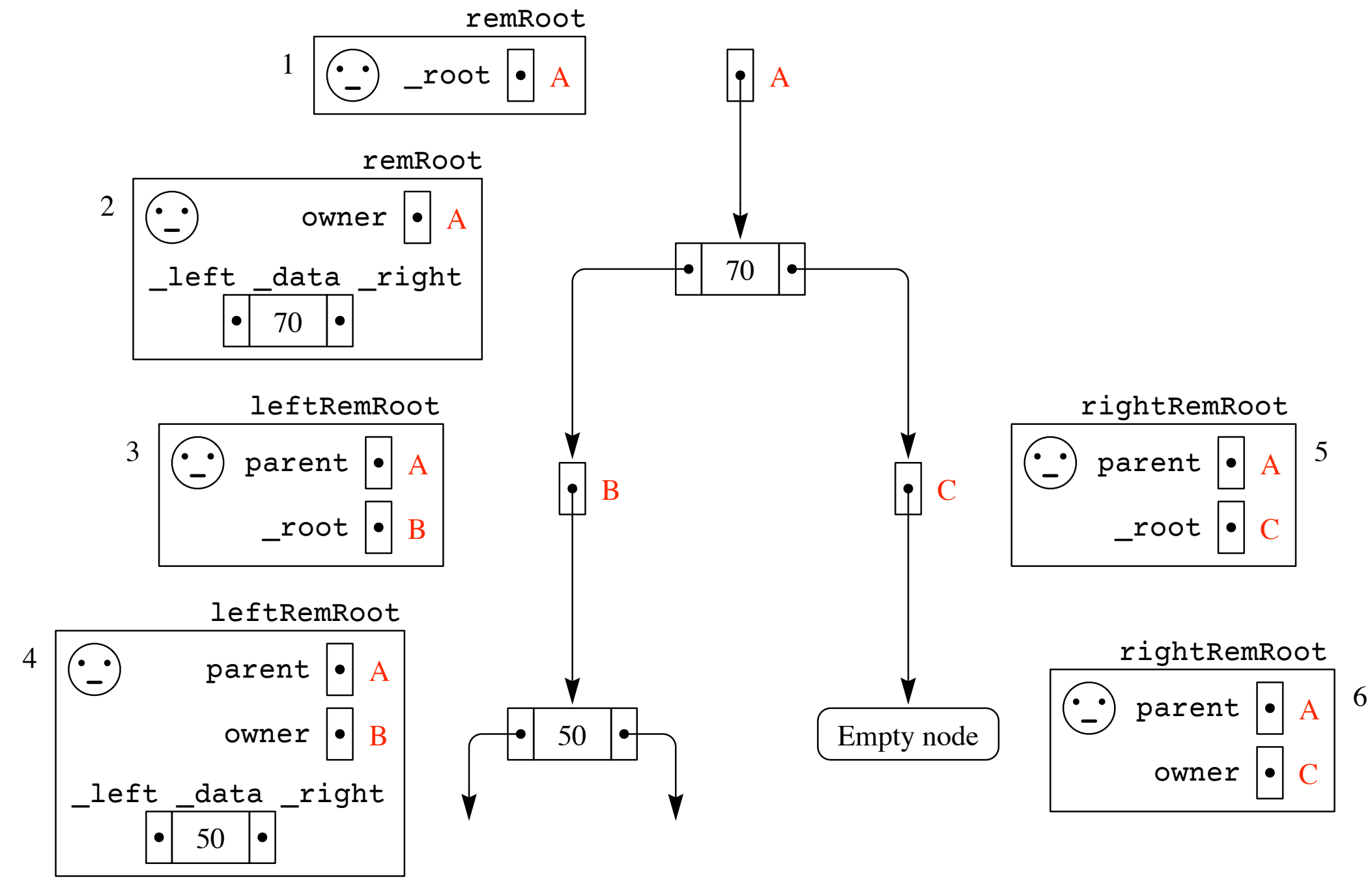
1. Tree `remRoot()` calls NENode `remRoot()`
2. NENode `remRoot()` calls left tree `leftRemRoot()`



3. Tree `leftRemRoot()` calls NENode `leftRemRoot()`
4. NENode `leftRemRoot()` calls parent's right `rightRemRoot()`

5. Tree `rightRemRoot()` calls MTNode `rightRemRoot()`
6. MTNode `rightRemRoot()` removes the root

(b) Call sequence of `remRoot()` for a nonempty left subtree and an empty right subtree.


remRoot
1
_root
A
A
remRoot
2
owner
A
_left _data _right
70
70
leftRemRoot
3
parent
A
_root
B
B
C
rightRemRoot
5
parent
A
_root
C
leftRemRoot
4
parent
A
owner
B
_left _data _right
50
50
Empty node
rightRemRoot
6
parent
A
owner
C

The signature mechanism

- A plugin is a subclass of an abstract visitor.
- It is free to add any number of attributes of any type that are not present in its superclass.
- The attributes in a visitor correspond to the parameters in a non-plugin method.

Visitor translation techniques

- Input parameters in the native method are attributes in the visitor initialized in the constructor for the visitor.
- Output parameters in the native method are reference variable attributes in the visitor.
- Nonvoid methods in the native class require an additional method named `result()` that returns the same type.

`BiTCSVisEmptyVis`

The `BiTreeCS` version of `isEmpty()` is non-void.

Therefore, the visitor version provides a `result()` method.

To use a visitor for a nonvoid method:

- Declare a local visitor.
- Pass it as a parameter in the data structure's `accept()` method.
- Call the visitor's `result()` method to get the result of the computation.

```
// ========= BiTCSVisEmptyVis =========
template<class T>
class BiTCSVisEmptyVis : public ABiTreeCSVVis<T> {
private:
    bool _result; // Output result.

public:
    // ========= Constructor =========
    BiTCSVisEmptyVis() {
    }

    // ========= visit =========
    void emptyCase(BiTreeCSV<T> &host) override {
        _result = true;
    }

    void nonEmptyCase(BiTreeCSV<T> &host) override {
        _result = false;
    }
```

```
    // ========= result =========
    // Pre: This visitor has been accepted by a host tree.
    // Post: true is returned if the host tree is empty;
    // otherwise, false is returned.
    bool result() const {
        return _result;
    }
};

// Global function for convenience
template<class T>
bool isEmpty(BiTreeCSV<T> const &tree) {
    BiTCSVisEmptyVis<T> isEmptyVis;
    tree.accept(isEmptyVis);
    return isEmptyVis.result();
}
```

General translation rules

- Use `host.left()` in a `BiTreeCSV` visitor to correspond to `_left` in a `BiTreeCS` nonempty node.
- Use `host.root()` in a `BiTreeCSV` visitor to correspond to `_data` in a `BiTreeCS` nonempty node.
- Use `host.right()` in a `BiTreeCSV` visitor to correspond to `_right` in a `BiTreeCS` nonempty node.

`BiTCSVpreOrderVis`

The `BiTreeCS` version of `preOrder()` is void.

Therefore, the visitor version does not provide a `result()` method.

The `BiTreeCS` version has output parameter `os`, so the visitor version has reference variable attribute `_os`.

```
// ========= BiTCSVpreOrderVis =========
template<class T>
class BiTCSVpreOrderVis : public ABiTreeCSVVis<T> {
private:
    ostream &_os; // Input parameter.

public:
    // ========= Constructor =========
    BiTCSVpreOrderVis(ostream &os):
        _os(os) {
    }

    // ========= visit =========
    // Pre: This visitor has been accepted by a host tree.
    // Post: A preorder representation of this tree is sent to os.
    void emptyCase(BiTreeCSV<T> &host) override {
    }

    void nonEmptyCase(BiTreeCSV<T> &host) override {
        _os << host.root() << "  ";
        host.left()->accept(*this);
        host.right()->accept(*this);
    }
};
```

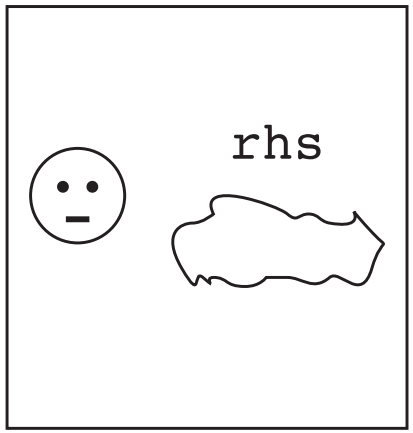
```
// Global function for convenience
template<class T>
void preOrder(ostream &os, BiTreeCSV<T> const &tree) {
    BiTCSVpreOrderVis<T> preOrderVis(os);
    tree.accept(preOrderVis);
}
```

```
// ========= BiTCSVnumNodesVis =========
template<class T>
class BiTCSVnumNodesVis : public ABiTreeCSVVis<T> {
private:
    int _result; // Output result.

public:
    // ========= Constructor =========
    BiTCSVnumNodesVis():
        _result(0) {
    }

    // ========= visit =========
    void emptyCase(BiTreeCSV<T> &host) override {
    }

    void nonEmptyCase(BiTreeCSV<T> &host) override {
        host.left()->accept(*this);
        host.right()->accept(*this);
        _result++;
    }
```

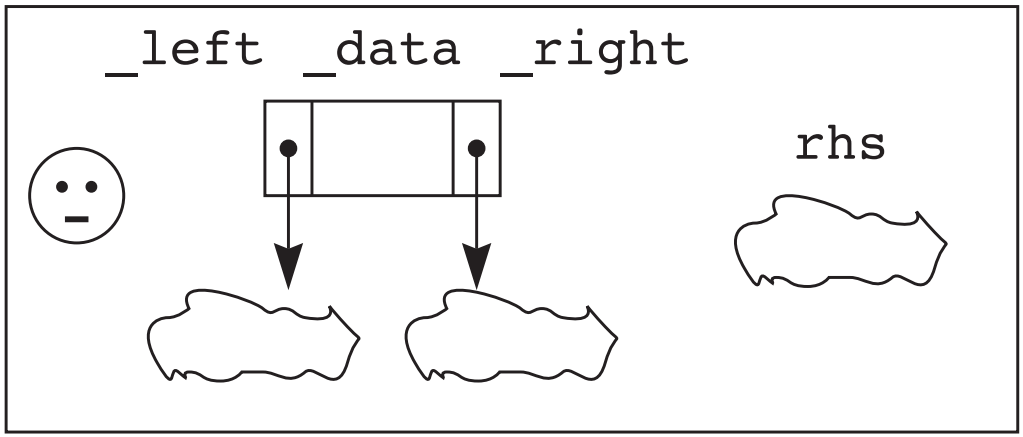
```
    // ========= result =========
    // Pre: This visitor has been accepted by a host tree.
    // Post: The number of nodes of the host tree is returned.
    int result() const {
        return _result;
    }
};

// Global function for convenience
template<class T>
int numNodes(BiTreeCSV<T> const &tree) {
    BiTCSVnumNodesVis<T> numNodesVis;
    tree.accept(numNodesVis);
    return numNodesVis.result();
}
```

```
// ========= BiTCSVequalsVis =========
template<class T>
class BiTCSVequalsVis : public ABiTreeCSVVis<T> {
private:
    BiTreeCSV<T> const &_rhs; // Input parameter.
    bool _result; // Output result.

public:
    // ========= Constructor =========
    BiTCSVequalsVis(BiTreeCSV<T> const &rhs):
        _rhs(rhs) {
    }

    // ========= visit =========
    void emptyCase(BiTreeCSV<T> &host) override {
        cerr << "BiTCSVequalsVis: Exercise for the student."
             << endl;
        throw -1;
    }

    void nonEmptyCase(BiTreeCSV<T> &host) override {
        cerr << "BiTCSVequalsVis: Exercise for the student."
             << endl;
        throw -1;
    }
```

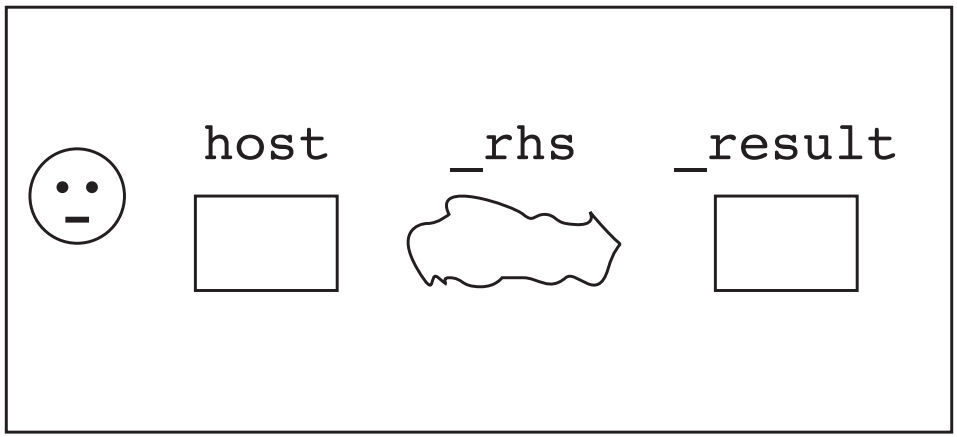
```
    // ========= result =========
    // Pre: This visitor has been accepted by a host tree.
    // Post: true is returned if the host tree is equal to rhs;
    // otherwise, false is returned.
    // Two trees are equal if they contain the same number of
    // equal elements with the same shape.
    bool result() const {
        cerr << "BiTCSVequalsVis: Exercise for the student."
             << endl;
        throw -1;
    }
};

// Global function for convenience.
// ========= operator== =========
template<class T>
bool operator==(BiTreeCSV<T> const &lhs, BiTreeCSV<T> const &rhs) {
    cerr << "operator==: Exercise for the student." << endl;
    throw -1;
}
```



(a) The environment of a `BiTreeCS` empty node.



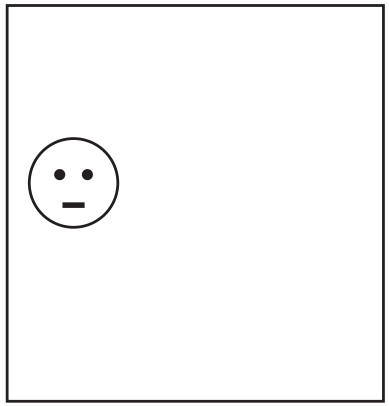
(b) The environment of a `BiTreeCS` nonempty node.



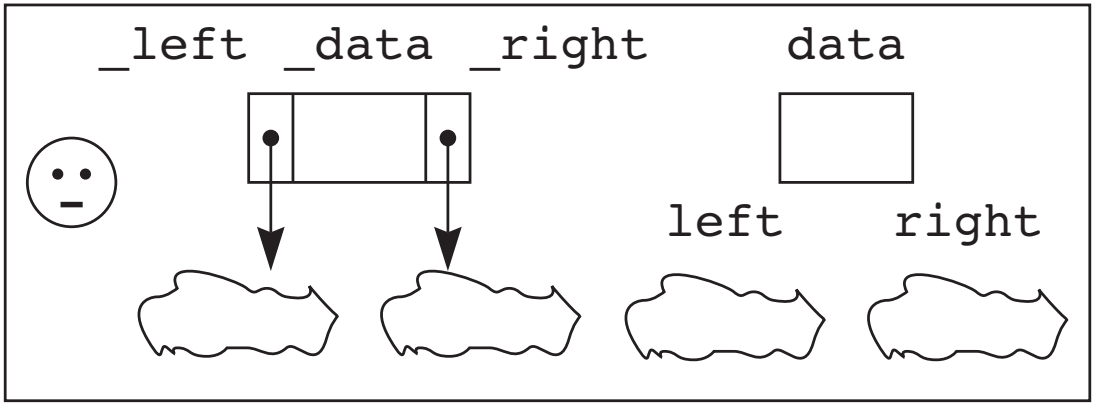
(c) The environment of a `BiTreeCSV` helper visitor.

```
// ========= BiTCSVequalsHelperVis =========
template<class T>
class BiTCSVequalsHelperVis : public ABiTreeCSVVis<T> {
private:
    T const &_data; // Input parameter.
    BiTreeCSV<T> const &_left; // Input parameter.
    BiTreeCSV<T> const &_right; // Input parameter.
    bool _result; // Output result.

public:
    // ========= Constructor =========
    BiTCSVequalsHelperVis
    (T const &data, BiTreeCSV<T> const &left, BiTreeCSV<T> const &right):
        _data(data), _left(left), _right(right) {
    }

    // ========= visit =========
    void emptyCase(BiTreeCSV<T> &host) override {
        cerr << "BiTCSVequalsHelperVis: Exercise for the student."
             << endl;
        throw -1;
    }

    void nonEmptyCase(BiTreeCSV<T> &host) override {
        cerr << "BiTCSVequalsHelperVis: Exercise for the student."
             << endl;
        throw -1;
    }
```

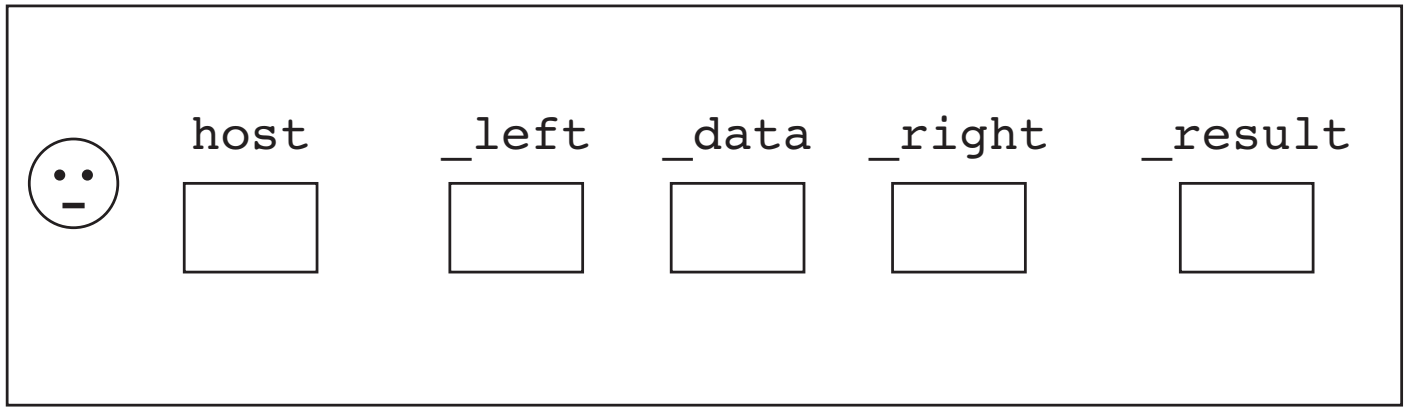
```
    // ========= result =========
    // Pre: This visitor has been accepted by a host tree.
    // Post: true is returned if root equals host.root(),
    // left equals host.left(), and right equals host.right();
    // otherwise, false is returned.
    bool result() const {
        cerr << "BiTCSVequalsHelperVis: Exercise for the student."
             << endl;
        throw -1;
    }
};
```



(a) The environment of a `BiTreeCS` helper empty node.

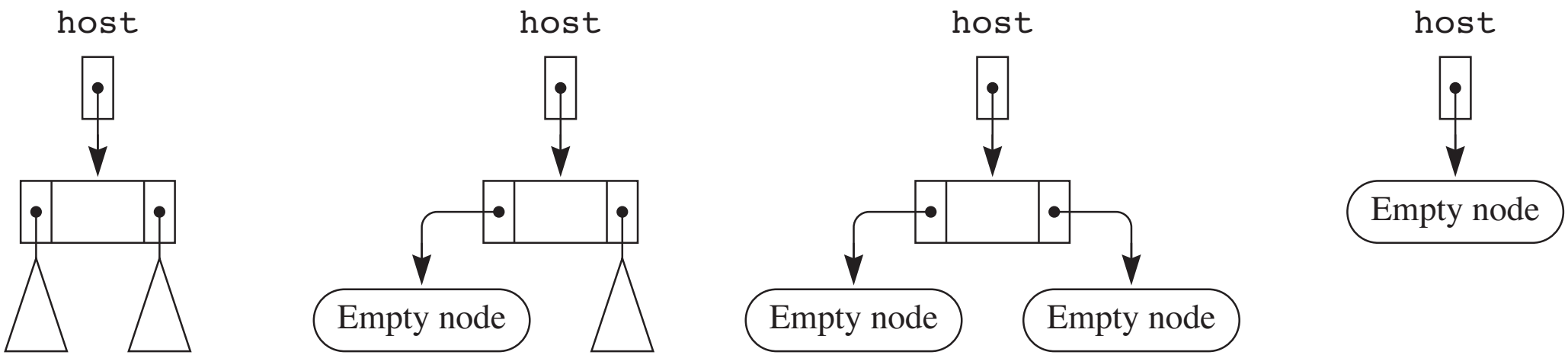


(b) The environment of a `BiTreeCS` helper nonempty node.



(c) The environment of a `BiTreeCSV` helper visitor.

The `clear()` operation for `BiTreeCSV`



(a) Original. **(b)** Clear left. **(c)** Clear right. **(d)** Remove root.