The Composite State Binary Tree BiTreeCS

The Composite State Binary Tree BiTreeCS

In a nonempty node

- `_left` is a binary tree
- `_right` is a binary tree

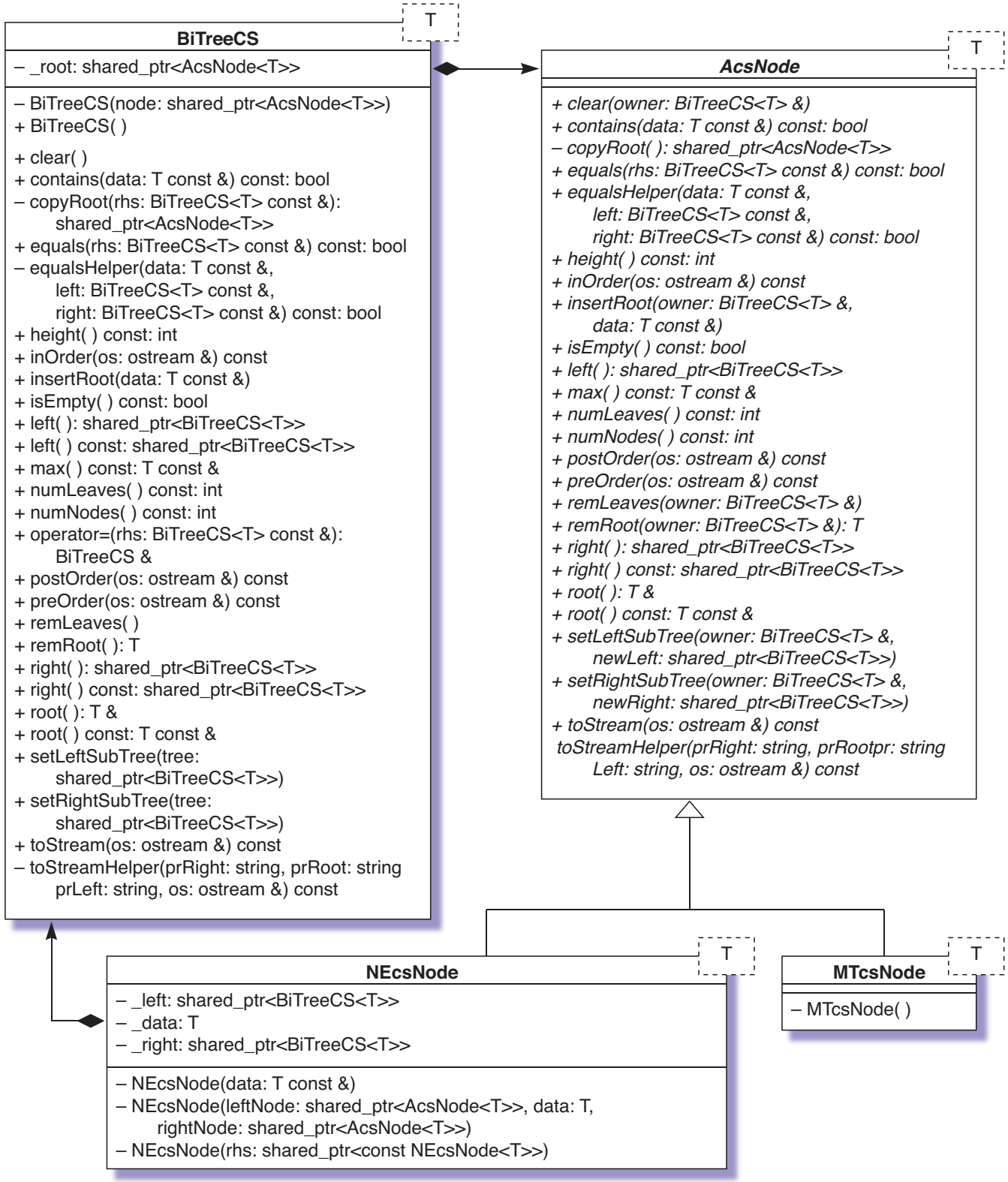

T
BiTreeCS
– _root: shared_ptr<AcsNode<T>>
– BiTreeCS(node: shared_ptr<AcsNode<T>>)
+ BiTreeCS( )
+ clear( )
+ contains(data: T const &) const: bool
– copyRoot(rhs: BiTreeCS<T> const &):
shared_ptr<AcsNode<T>>
+ equals(rhs: BiTreeCS<T> const &) const: bool
– equalsHelper(data: T const &,
left: BiTreeCS<T> const &,
right: BiTreeCS<T> const &) const: bool
+ height( ) const: int
+ inOrder(os: ostream &) const
+ insertRoot(data: T const &)
+ isEmpty( ) const: bool
+ left( ): shared_ptr<BiTreeCS<T>>
+ left( ) const: shared_ptr<BiTreeCS<T>>
+ max( ) const: T const &
+ numLeaves( ) const: int
+ numNodes( ) const: int
+ operator=(rhs: BiTreeCS<T> const &):
BiTreeCS &
+ postOrder(os: ostream &) const
+ preOrder(os: ostream &) const
+ remLeaves( )
+ remRoot( ): T
+ right( ): shared_ptr<BiTreeCS<T>>
+ right( ) const: shared_ptr<BiTreeCS<T>>
+ root( ): T &
+ root( ) const: T const &
+ setLeftSubTree(tree:
shared_ptr<BiTreeCS<T>>)
+ setRightSubTree(tree:
shared_ptr<BiTreeCS<T>>)
+ toStream(os: ostream &) const
– toStreamHelper(prRight: string, prRoot: string
prLeft: string, os: ostream &) const
T
AcsNode
+ clear(owner: BiTreeCS<T> &)
+ contains(data: T const &) const: bool
– copyRoot( ): shared_ptr<AcsNode<T>>
+ equals(rhs: BiTreeCS<T> const &) const: bool
+ equalsHelper(data: T const &,
left: BiTreeCS<T> const &,
right: BiTreeCS<T> const &) const: bool
+ height( ) const: int
+ inOrder(os: ostream &) const
+ insertRoot(owner: BiTreeCS<T> &,
data: T const &)
+ isEmpty( ) const: bool
+ left( ): shared_ptr<BiTreeCS<T>>
+ max( ) const: T const &
+ numLeaves( ) const: int
+ numNodes( ) const: int
+ postOrder(os: ostream &) const
+ preOrder(os: ostream &) const
+ remLeaves(owner: BiTreeCS<T> &)
+ remRoot(owner: BiTreeCS<T> &): T
+ right( ): shared_ptr<BiTreeCS<T>>
+ right( ) const: shared_ptr<BiTreeCS<T>>
+ root( ): T &
+ root( ) const: T const &
+ setLeftSubTree(owner: BiTreeCS<T> &,
newLeft: shared_ptr<BiTreeCS<T>>)
+ setRightSubTree(owner: BiTreeCS<T> &,
newRight: shared_ptr<BiTreeCS<T>>)
+ toStream(os: ostream &) const
toStreamHelper(prRight: string, prRootpr: string
Left: string, os: ostream &) const
T
NEcsNode
– _left: shared_ptr<BiTreeCS<T>>
– _data: T
– _right: shared_ptr<BiTreeCS<T>>
– NEcsNode(data: T const &)
– NEcsNode(leftNode: shared_ptr<AcsNode<T>>, data: T,
rightNode: shared_ptr<AcsNode<T>>)
– NEcsNode(rhs: shared_ptr<const NEcsNode<T>>)
T
MTcsNode
– MTcsNode( )

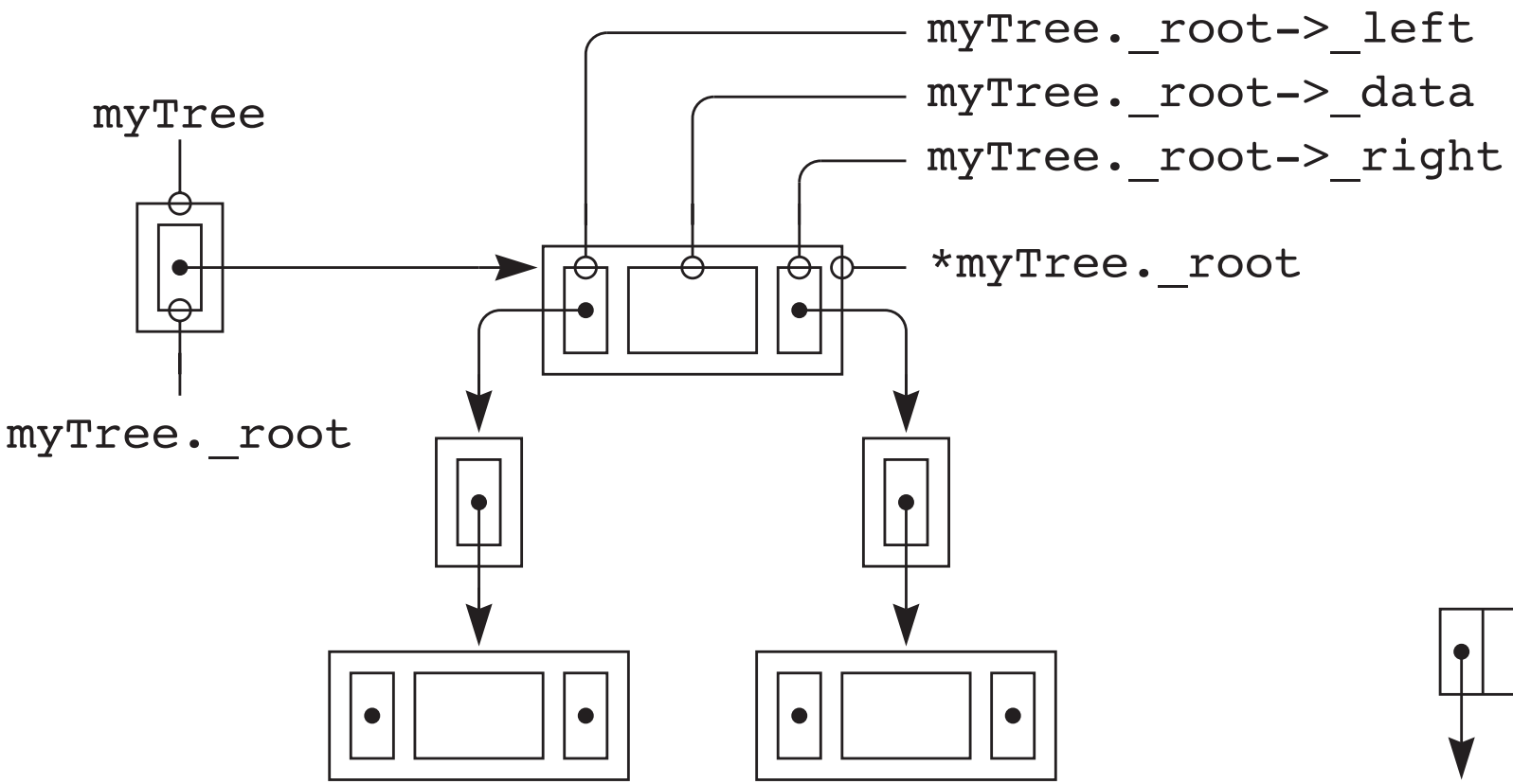
Figure 8.24

```
template<class T> class BiTreeCS {
private:
     shared_ptr<AcsNode<T>> _root;
```

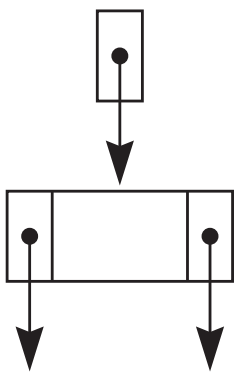
```
template<class T> class NEcsNode : public AcsNode<T> {
private:
    shared_ptr<BiTreeCS<T>> _left;
    T _data;
    shared_ptr<BiTreeCS<T>> _right;
```

```
template<class T> class BiTreeCS {
private:
     shared_ptr<AcsNode<T>> _root;
```

```
template<class T> class NEcsNode : public AcsNode<T> {
private:
     shared_ptr<BiTreeCS<T>> _left;
     T _data;
     shared_ptr<BiTreeCS<T>> _right;
```



(a) A detailed rendering of a binary tree and a node.



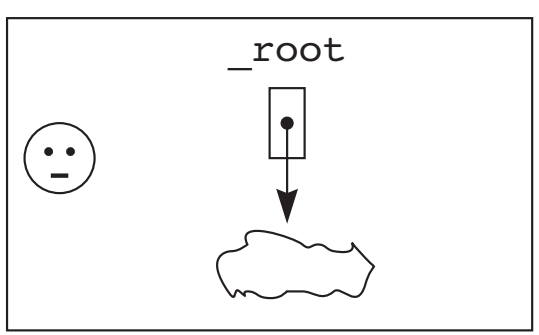
(b) An abbreviated rendering.

Methods for output and characterization

```
// ========= preOrder =========
// Post: A preorder representation of this tree
// is sent to os.

template<class T>
void BiTreeCS<T>::preOrder(ostream &os) const {
    _root->preOrder(os);
}

template<class T>
void MTcsNode<T>::preOrder(ostream &os) const {
}

template<class T>
void NEcsNode<T>::preOrder(ostream &os) const {
    os << _data << "  ";
    _left->preOrder(os);
    _right->preOrder(os);
}
```

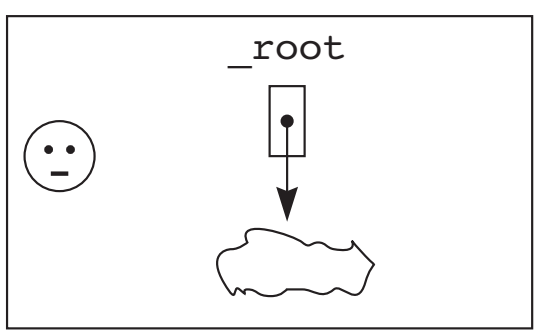
```
// ========= isEmpty =========
// Post: true is returned if this tree is empty;
// otherwise, false is returned.

template<class T>
bool BiTreeCS<T>::isEmpty() const {
    return _root->isEmpty();
}

template<class T>
bool MTcsNode<T>::isEmpty() const {
    return true;
}

template<class T>
bool NEcsNode<T>::isEmpty() const {
    return false;
}
```

```
// ========= isEmpty =========
// Post: true is returned if this tree is empty;
// otherwise, false is returned.

template<class T>
bool BiTreeCS<T>::isEmpty() const {
    return _root->isEmpty();
}

template<class T>
bool MTcsNode<T>::isEmpty() const {
    return true;
}

template<class T>
bool NEcsNode<T>::isEmpty() const {
    return false;
}
```

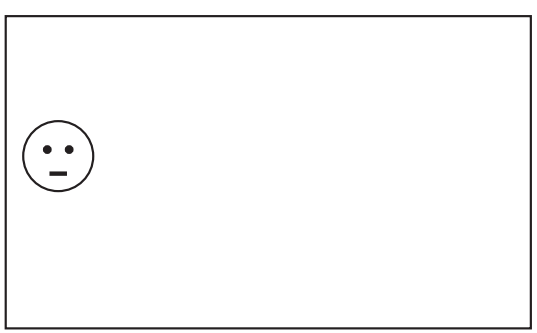


(a) The environment of a binary tree.

```
// ========= isEmpty =========
// Post: true is returned if this tree is empty;
// otherwise, false is returned.

template<class T>
bool BiTreeCS<T>::isEmpty() const {
    return _root->isEmpty();
}

template<class T>
bool MTcsNode<T>::isEmpty() const {
    return true;
}

template<class T>
bool NEcsNode<T>::isEmpty() const {
    return false;
}
```

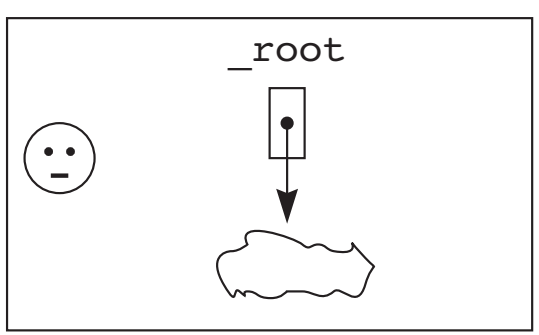


(a) The environment of a binary tree.

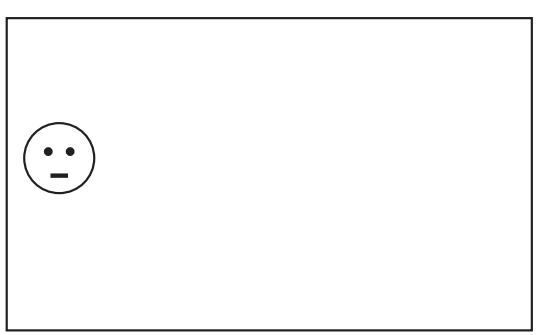


(b) The environment of an empty node.

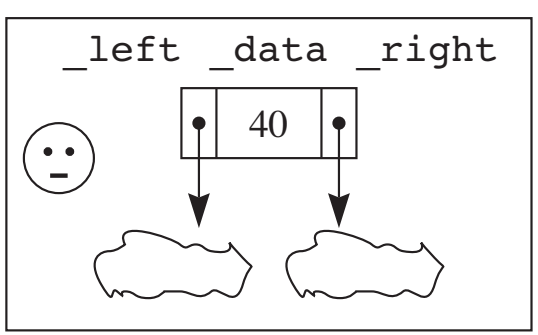
```
// ========= isEmpty =========
// Post: true is returned if this tree is empty;
// otherwise, false is returned.

template<class T>
bool BiTreeCS<T>::isEmpty() const {
    return _root->isEmpty();
}

template<class T>
bool MTcsNode<T>::isEmpty() const {
    return true;
}

template<class T>
bool NEcsNode<T>::isEmpty() const {
    return false;
}
```



(a) The environment of a binary tree.



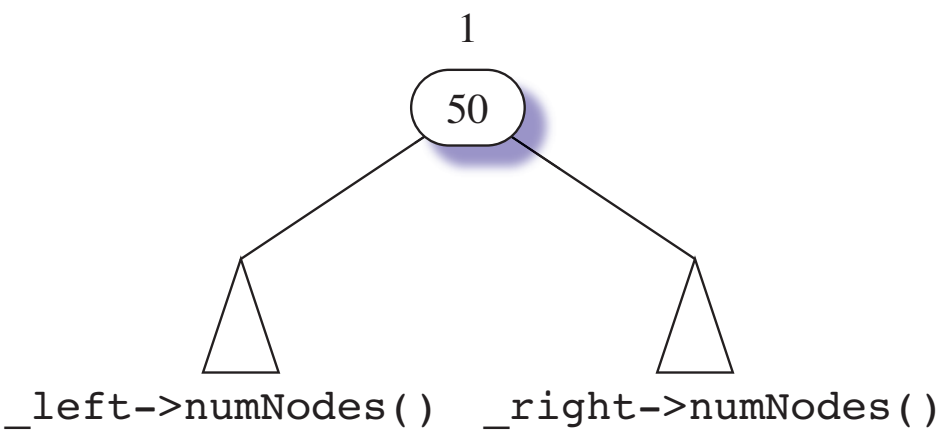
(b) The environment of an empty node.



(c) The environment of a nonempty node.

```
// ========= left =========
// Pre: This tree is not empty.
// Post: A shared_ptr to the left child of this tree is returned.

template<class T>
shared_ptr<BiTreeCS<T>> BiTreeCS<T>::left() {
    return _root->left();
};

template<class T>
shared_ptr<BiTreeCS<T>> MTcsNode<T>::left() {
    cerr << "left precondition violated: "
         << "An empty tree has has no left subtree."
         << endl;
    throw -1;
}

template<class T>
shared_ptr<BiTreeCS<T>> NEcsNode<T>::left() {
    return _left;
}
```

```
// ========= numNodes =========
// Post: The number of nodes of this tree is returned.

template<class T>
int BiTreeCS<T>::numNodes() const {
    return _root->numNodes();
}

template<class T>
int MTcsNode<T>::numNodes() const {
    return 0;
}

template<class T>
int NEcsNode<T>::numNodes() const {
    return 1 + _left->numNodes() + _right->numNodes();
}
```

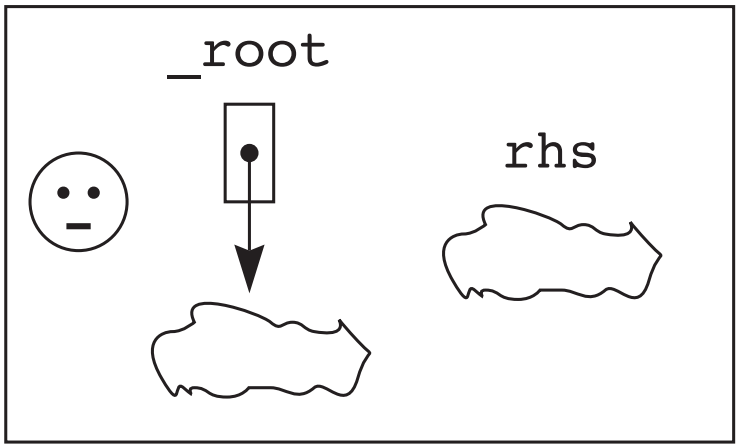


```
// ========= max =========
// Pre: This tree is not empty.
// Post: The maximum element of this tree is returned.

template<class T>
T const &BiTreeCS<T>::max() const {
    return _root->max();
}

template<class T>
T const &MTcsNode<T>::max() const {
    cerr << "max precondition violated: "
         << "An empty tree has no maximum." << endl;
    throw -1;
}

template<class T>
T const &NEcsNode<T>::max() const {
    T const &leftMax = (_left->isEmpty()) ? _data : _left->max();
    T const &rightMax = (_right->isEmpty()) ? _data : _right->max();
    return (leftMax > rightMax) ?
        ((leftMax > _data) ? leftMax : _data) :
        ((rightMax > _data) ? rightMax: _data);
}
```

```
// ========= equals =========
// Post: true is returned if this tree equals tree rhs;
// otherwise, false is returned.
// Two trees are equal if they contain the same number
// of equal elements with the same shape.

template<class T>
bool BiTreeCS<T>::equals(BiTreeCS<T> const &rhs) const {
    cerr << "BiTreeCS<T>::equals: Exercise for the student." << endl;
    throw -1;
}

template<class T>
bool MTcsNode<T>::equals(BiTreeCS<T> const &rhs) const {
    cerr << "MTcsNode<T>::equals: Exercise for the student." << endl;
    throw -1;
}

template<class T>
bool NEcsNode<T>::equals(BiTreeCS<T> const &rhs) const {
    cerr << "NEcsNode<T>::equals: Exercise for the student." << endl;
    throw -1;
}
```

```
template<class T>
bool BiTreeCS<T>::equals(BiTreeCS<T> const &rhs) const {
    cerr << "BiTreeCS<T>::equals: Exercise for the student." << endl;
    throw -1;
}
```

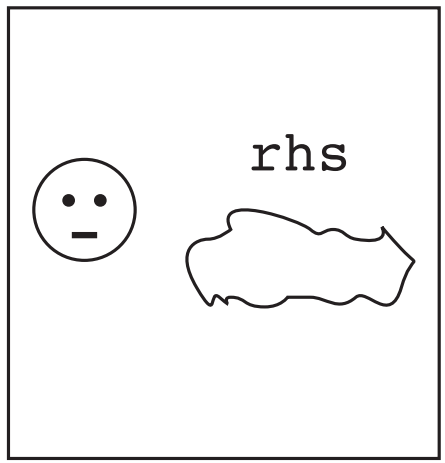
```
template<class T>
bool BiTreeCS<T>::equals(BiTreeCS<T> const &rhs) const {
    cerr << "BiTreeCS<T>::equals: Exercise for the student." << endl;
    throw -1;
}
```



(a) The environment of a binary tree.

```
template<class T>
bool MTcsNode<T>::equals(BiTreeCS<T> const &rhs) const {
    cerr << "MTcsNode<T>::equals: Exercise for the student." << endl;
    throw -1;
}
```

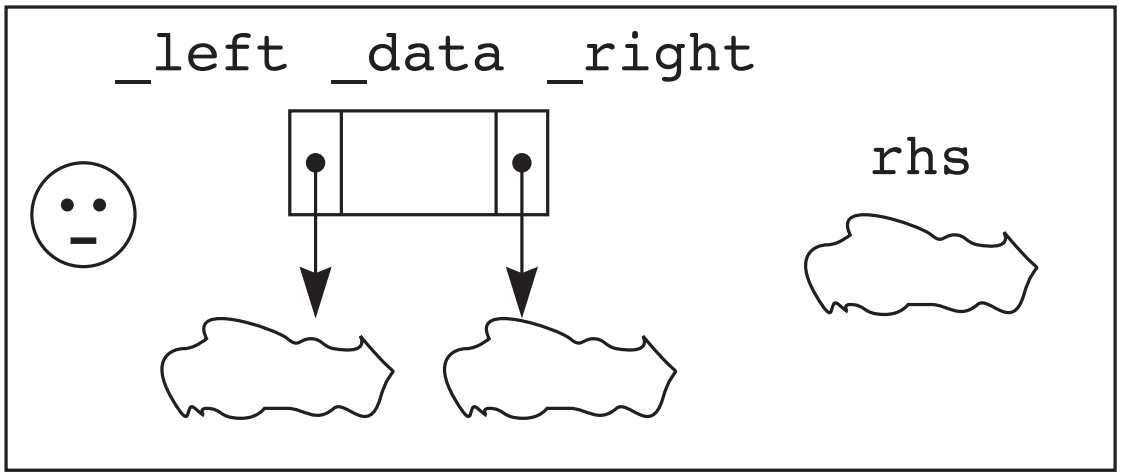
```
template<class T>
bool MTcsNode<T>::equals(BiTreeCS<T> const &rhs) const {
    cerr << "MTcsNode<T>::equals: Exercise for the student." << endl;
    throw -1;
}
```



(b) The environment of an empty node.

```
template<class T>
bool NEcsNode<T>::equals(BiTreeCS<T> const &rhs) const {
    cerr << "NEcsNode<T>::equals: Exercise for the student." << endl;
    throw -1;
}
```

```
template<class T>
bool NEcsNode<T>::equals(BiTreeCS<T> const &rhs) const {
    cerr << "NEcsNode<T>::equals: Exercise for the student." << endl;
    throw -1;
}
```

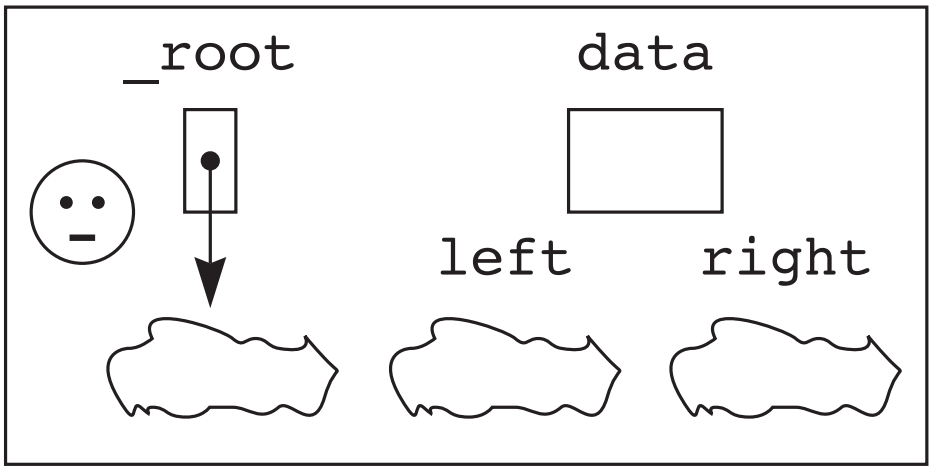


(c) The environment of a nonempty node.

```
// --------- equalsHelper ---------
// Post: true is returned if root equals this->root(),
// left equals this->left(), and right equals this->right();
// otherwise, false is returned.

template<class T>
bool BiTreeCS<T>::equalsHelper(T const &data,
    BiTreeCS<T> const &left, BiTreeCS<T> const &right) const {
    cerr << "BiTreeCS<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}

template<class T>
bool MTcsNode<T>::equalsHelper(T const &,
    BiTreeCS<T> const &, BiTreeCS<T> const &) const {
    cerr << "MTcsNode<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}

template<class T>
bool NEcsNode<T>::equalsHelper(T const &data,
    BiTreeCS<T> const &left, BiTreeCS<T> const &right) const {
    cerr << "NEcsNode<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}
```

```
// --------- equalsHelper ---------
template<class T>
bool BiTreeCS<T>::equalsHelper(T const &data,
    BiTreeCS<T> const &left, BiTreeCS<T> const &right) const {
    cerr << "BiTreeCS<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}
```

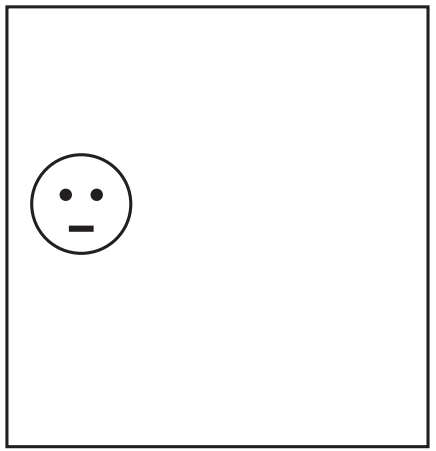
```
// --------- equalsHelper ---------
template<class T>
bool BiTreeCS<T>::equalsHelper(T const &data,
    BiTreeCS<T> const &left, BiTreeCS<T> const &right) const {
    cerr << "BiTreeCS<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}
```



(a) The environment of a binary tree.

```
template<class T>
bool MTcsNode<T>::equalsHelper(T const &,
    BiTreeCS<T> const &, BiTreeCS<T> const &) const {
    cerr << "MTcsNode<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}
```

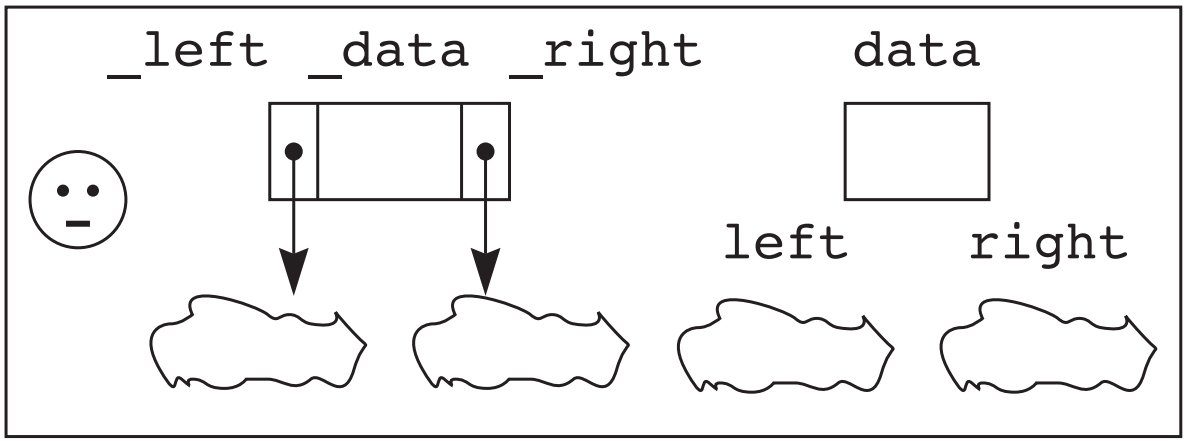
```
template<class T>
bool MTcsNode<T>::equalsHelper(T const &,
    BiTreeCS<T> const &, BiTreeCS<T> const &) const {
    cerr << "MTcsNode<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}
```



(b) The environment of an empty node.

```
template<class T>
bool NEcsNode<T>::equalsHelper(T const &data,
    BiTreeCS<T> const &left, BiTreeCS<T> const &right) const {
    cerr << "NEcsNode<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}
```

```
template<class T>
bool NEcsNode<T>::equalsHelper(T const &data,
    BiTreeCS<T> const &left, BiTreeCS<T> const &right) const {
    cerr << "NEcsNode<T>::equalsHelper: Exercise for the student." << endl;
    throw -1;
}
```

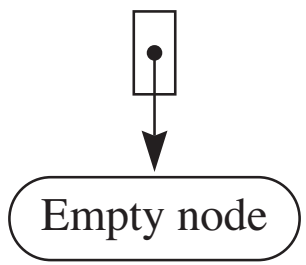


(c) The environment of a nonempty node.

Methods for construction and insertion

Two constructors for `BiTreeCS`

```
// ========== Constructors ==========
template<class T>
BiTreeCS<T>::BiTreeCS():
    _root(new MTcsNode<T>()) {
}

// Post: _root points to node with no allocation.
template<class T>
BiTreeCS<T>::BiTreeCS(shared_ptr<AcsNode<T>> node):
    _root(node) {
}
```



The empty tree created by `BiTreeCS()`.

Three constructors for NEcsNode

```
// Post: _data is data.
template<class T>
NEcsNode<T>::NEcsNode(T const &data):
    _left(make_shared<BiTreeCS<T>>()),
    _data(data),
    _right(make_shared<BiTreeCS<T>>()) {
}
```

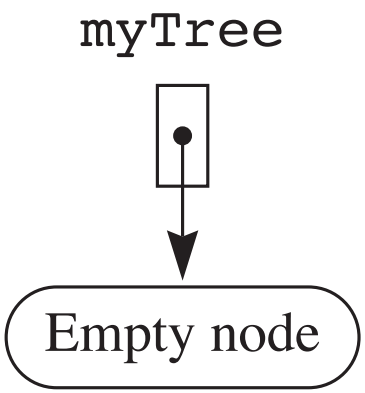
Three constructors for NEcsNode

```
// Post: _left->_root and _right->_root point to leftNode
// and rightNode, and _data is data.
template<class T>
NEcsNode<T>::NEcsNode(shared_ptr<AcsNode<T>> leftNode,
                      T data,
                      shared_ptr<AcsNode<T>> rightNode):
    NEcsNode(data) {
    _left->_root = leftNode;
    _right->_root = rightNode;
}
```

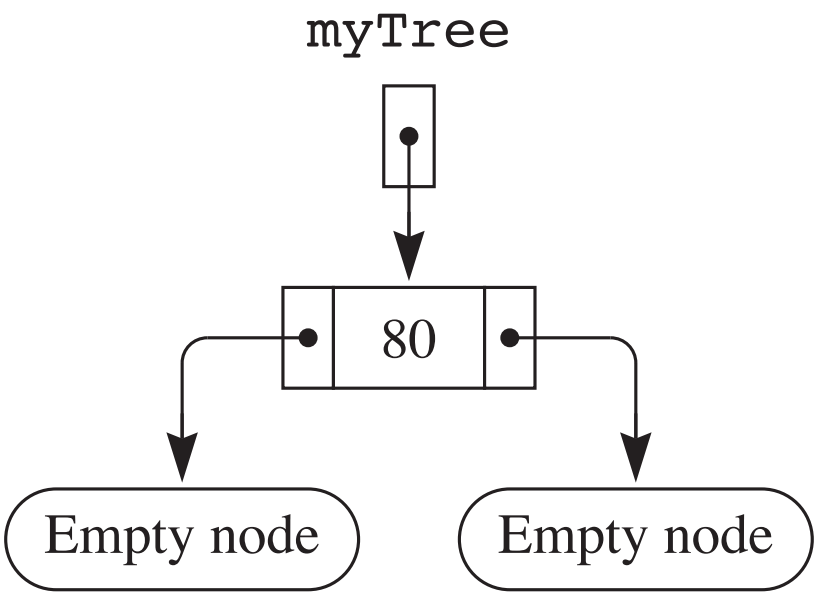
Three constructors for NEcsNode

```
// Post: _left is rhs->_left, _data is rhs->_data,
// and _right is rhs->_right.
template<class T>
NEcsNode<T>::NEcsNode(shared_ptr<const NEcsNode<T>> rhs):
    _left(rhs->_left),
    _data(rhs->_data),
    _right(rhs->_right) {
}
```

```
// ========= insertRoot =========
// Pre: This tree is empty.
// Post: This tree has one root node containing data.

template<class T>
void BiTreeCS<T>::insertRoot(T const &data) {
    _root->insertRoot(*this, data);
}

template<class T>
void MTcsNode<T>::insertRoot(BiTreeCS<T> &owner, T const &data) {
    owner._root.reset(new NEcsNode<T>(data));
}

template<class T>
void NEcsNode<T>::insertRoot(BiTreeCS<T> &, T const &) {
    cerr << "insertRoot precondition violated: "
         << "Cannot insert root into a non empty tree" << endl;
    throw -1;
}
```

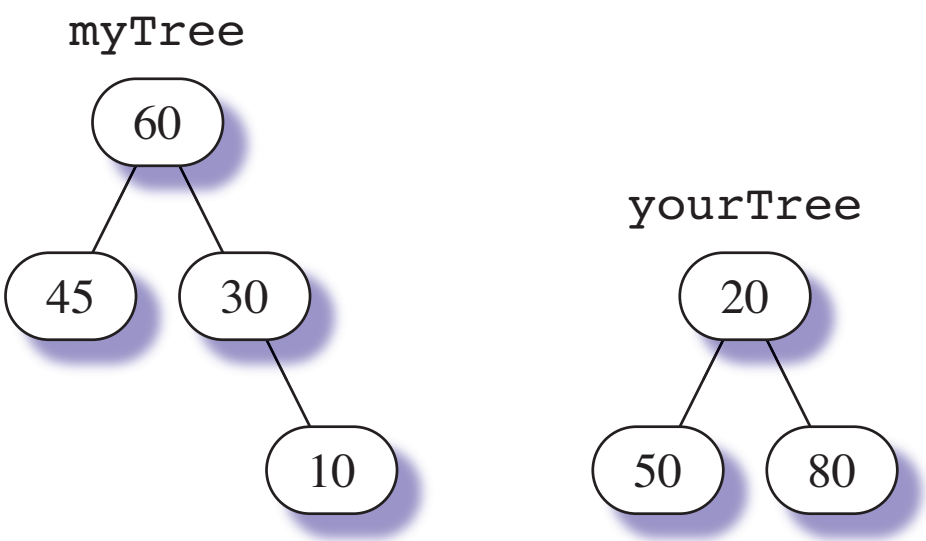


(a) Initial tree.

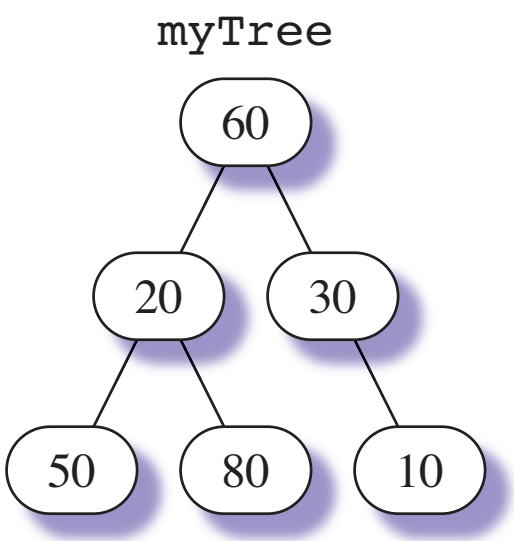


(b) `myTree.insertRoot(80);`

```
// ========= setLeftSubTree =========
// Pre: This tree is not empty.
// Post: The left subtree of this tree is replaced by newLeft.

template<class T>
void BiTreeCS<T>::setLeftSubTree(shared_ptr<BiTreeCS<T>> newLeft) {
    _root->setLeftSubTree(*this, newLeft);
}

template<class T>
void MTcsNode<T>::setLeftSubTree(BiTreeCS<T>&,
                                 shared_ptr<BiTreeCS<T>> ) {
    cerr << "setLeftSubTree precondition violated: "
         << "An empty list has no left subtree." << endl;
    throw -1;
}

template<class T>
void NEcsNode<T>::setLeftSubTree(BiTreeCS<T> &owner,
                                 shared_ptr<BiTreeCS<T>> newLeft) {
    _left = newLeft;
}
```



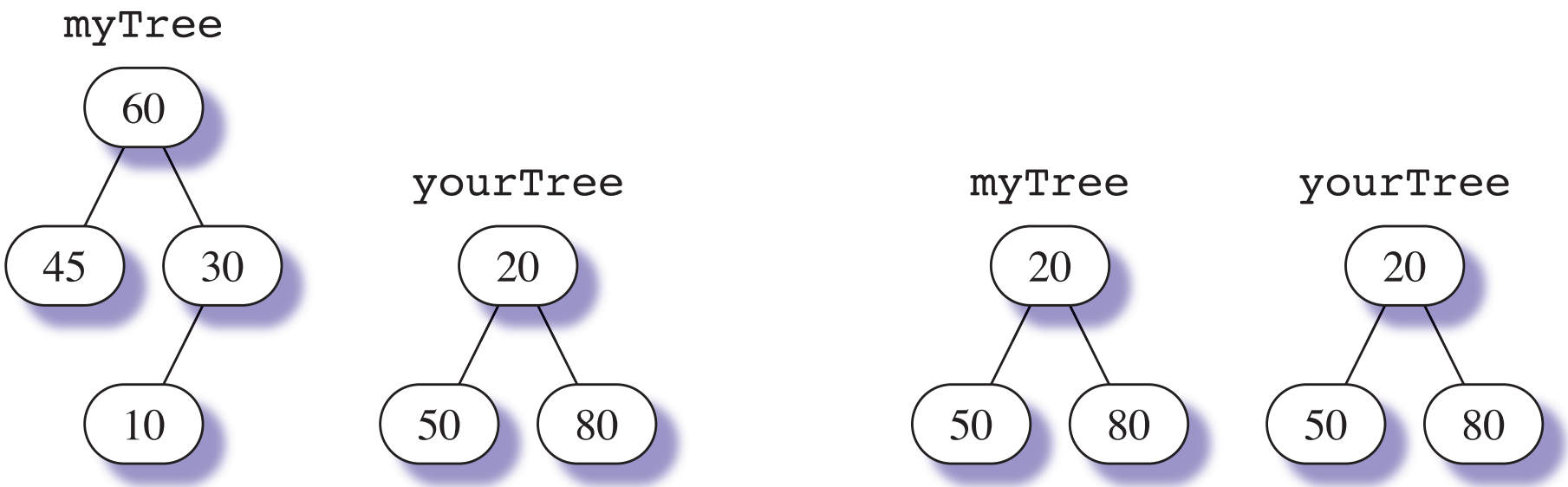
(a) Initial trees.



(b) `myTree.setLeftSubTree(yourTree);`

```
// ========= operator= =========
// Post: A deep copy of rhs is returned.

template<class T>
BiTreeCS<T> &BiTreeCS<T>::operator=(BiTreeCS<T> const &rhs) {
    if (this != &rhs) { // In case someone writes myTree = myTree;
        _root = copyRoot(rhs);
    }
    return *this;
}
```

```
// ========= copyRoot =========
// Post: A deep copy of the root of rhs is returned.

template<class T>
shared_ptr<AcsNode<T>> BiTreeCS<T>::copyRoot(BiTreeCS<T> const &rhs) {
    return rhs._root->copyRoot();
}

template<class T>
shared_ptr<AcsNode<T>> MTcsNode<T>::copyRoot() {
    return shared_ptr<MTcsNode<T>>(new MTcsNode<T>());
}

template<class T>
shared_ptr<AcsNode<T>> NEcsNode<T>::copyRoot() {
    return shared_ptr<NEcsNode<T>>(new NEcsNode<T>(
        _left->_root->copyRoot(),
        _data,
        _right->_root->copyRoot()));
}
```



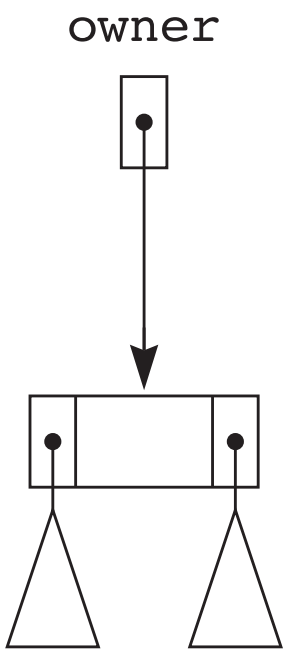
(a) Initial trees.

(b) `myTree = yourTree;`

Methods for removal

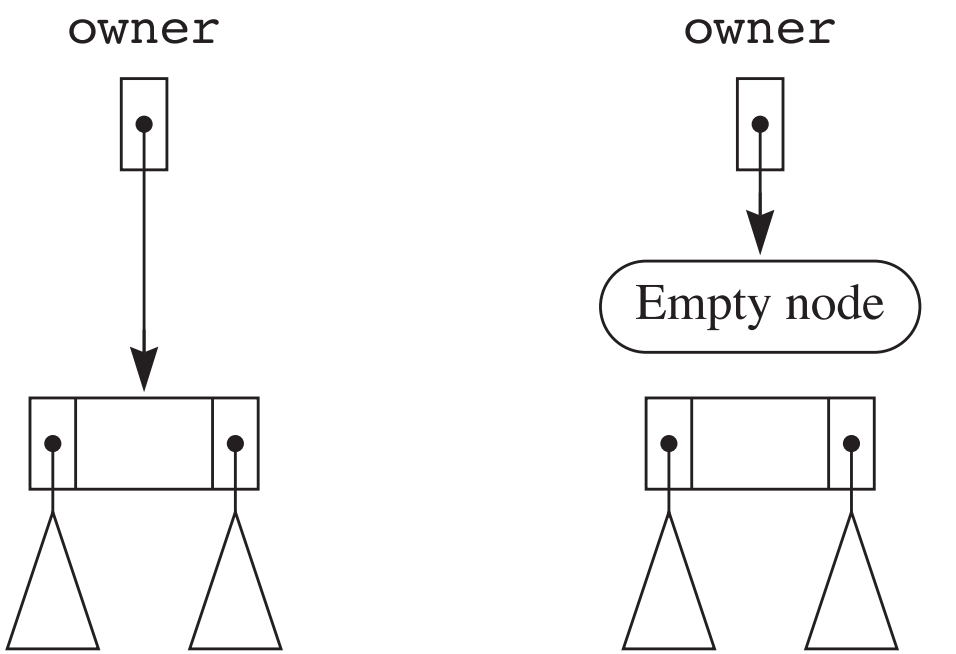
```
// ========= clear =========
// Post: This tree is cleared to the empty tree.

template<class T>
void BiTreeCS<T>::clear() {
    cerr << "BiTreeCS<T>::clear: Exercise for the student." << endl;
    throw -1;
}

template<class T>
void MTcsNode<T>::clear(BiTreeCS<T> &) {
    cerr << "MTcsNode<T>::clear: Exercise for the student." << endl;
    throw -1;
}

template<class T>
void NEcsNode<T>::clear(BiTreeCS<T> &owner) {
    cerr << "NEcsNode<T>::clear: Exercise for the student." << endl;
    throw -1;
}
```

The `clear()` operation for `BiTreeCS`



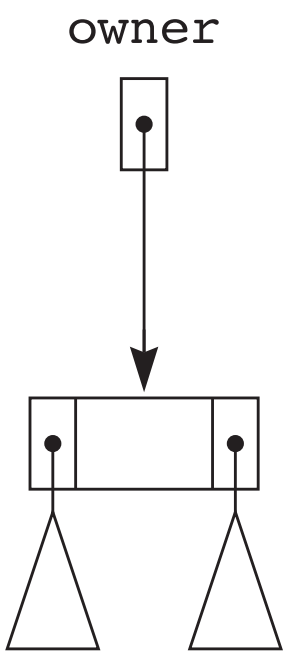
(a) Original.

The `clear()` operation for `BiTreeCS`

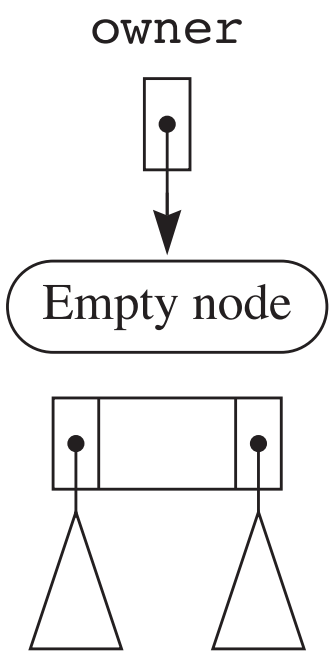


(a) Original. **(b)** Change owner.

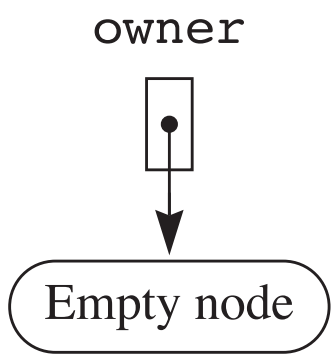
The `clear()` operation for `BiTreeCS`



(a) Original.

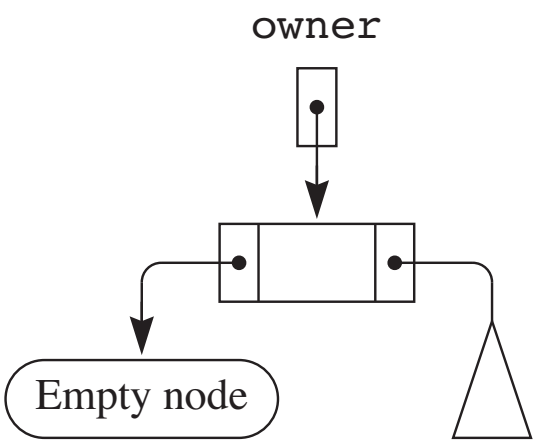


(b) Change owner.



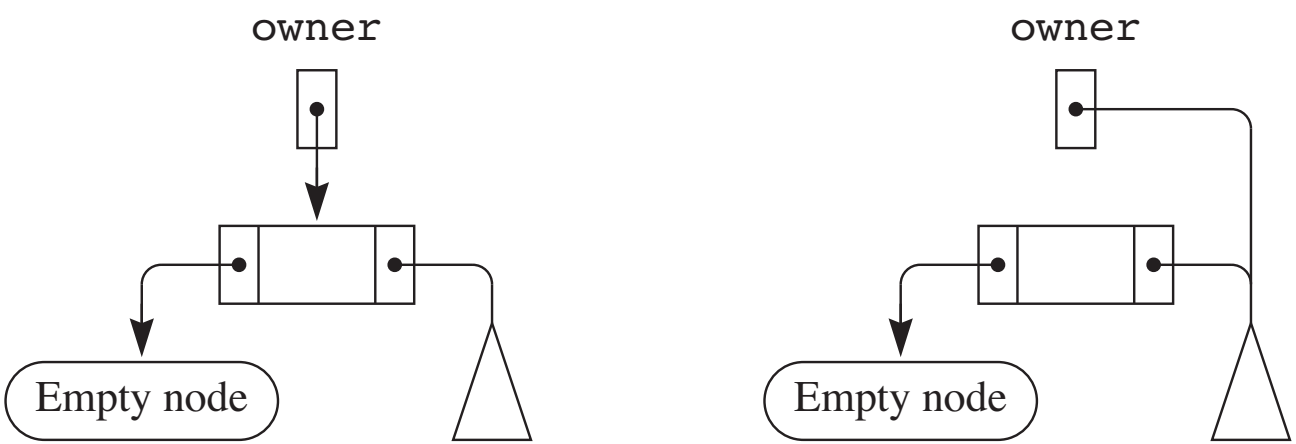
(c) Automatic garbage collection.

The `remRoot()` operation for `BiTreeCS`



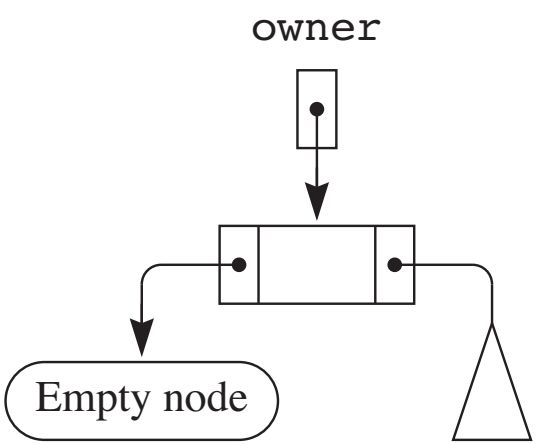
(a) Original.

The `remRoot()` operation for `BiTreeCS`

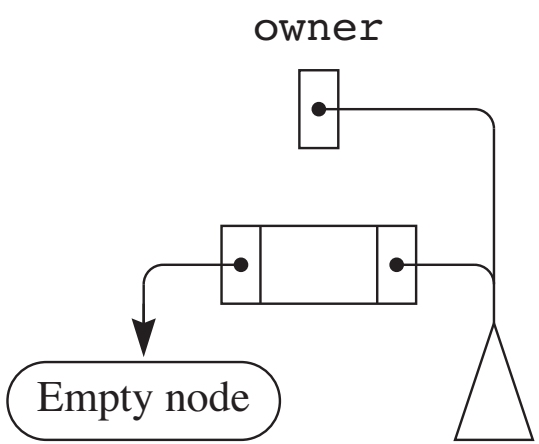


(a) Original. **(b)** Link owner.

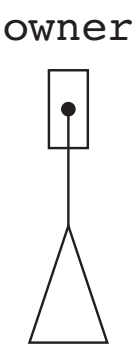
The `remRoot()` operation for `BiTreeCS`



(a) Original.



(b) Link owner.



(c) Automatic garbage collection