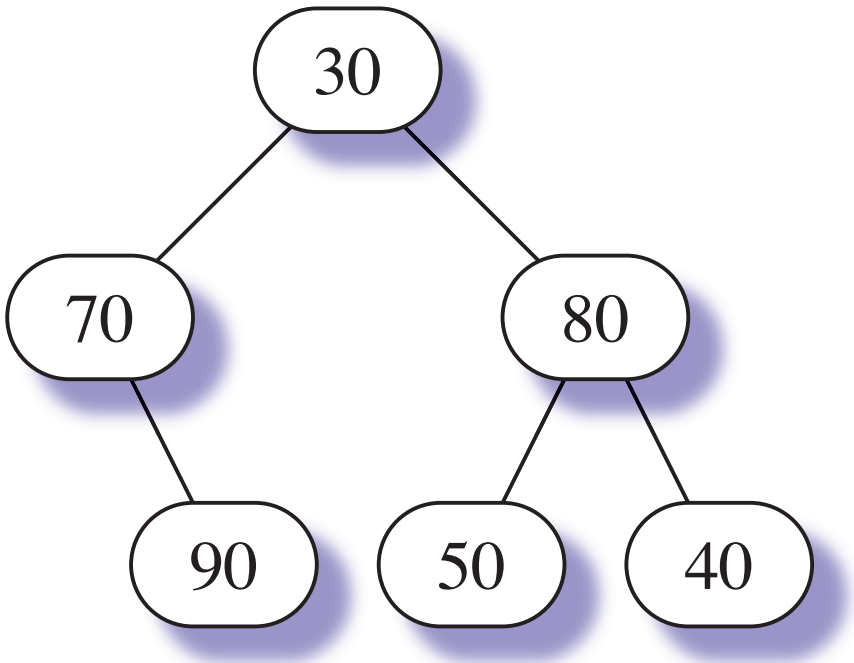
Binary Trees

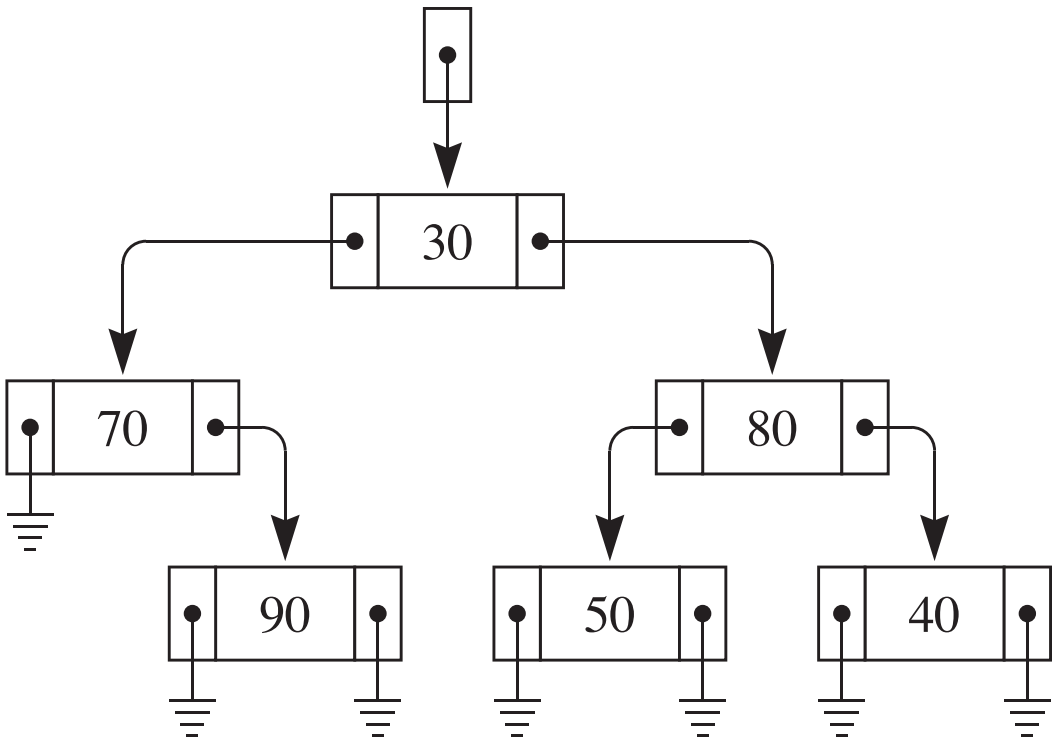
- `BiTreeL` Classic linked implementation
- `BiTreeCS` Composite and State patterns
- `BiTreeCSV` Composite, State, and Visitor patterns

The formal definition of a binary tree is recursive.

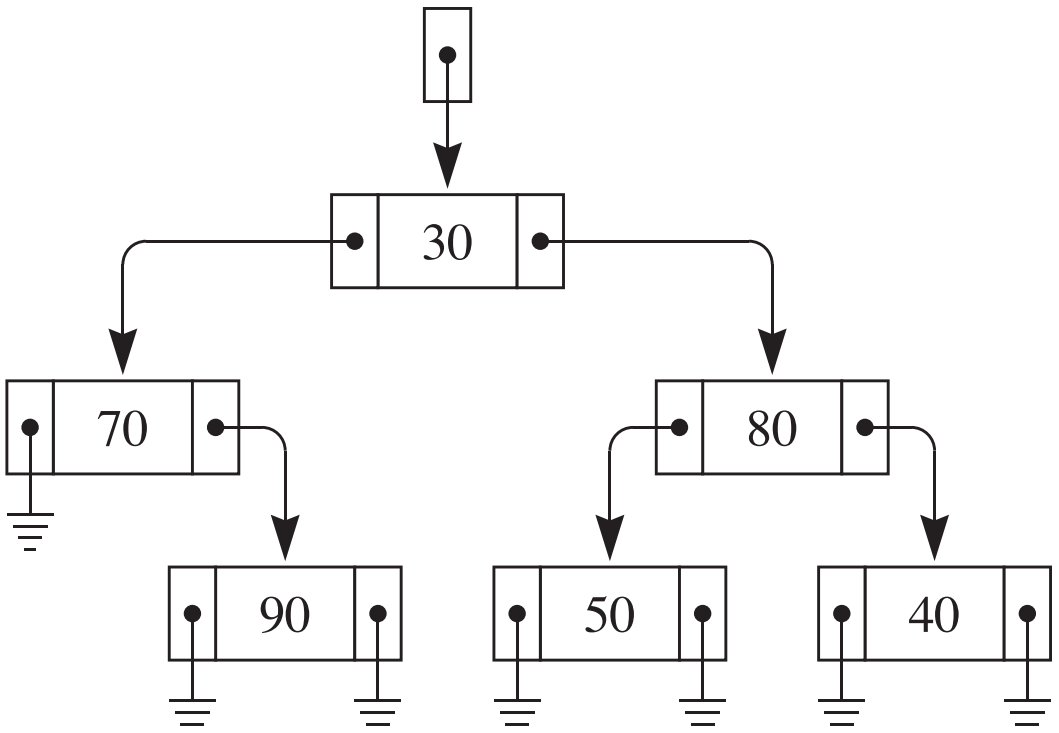
- An empty tree is a binary tree.
- A nonempty binary tree has three parts:
  - a left child, which is a binary tree,
  - a data value of some type T, and
  - a right child, which is a binary tree.



(a) An abstract binary tree.



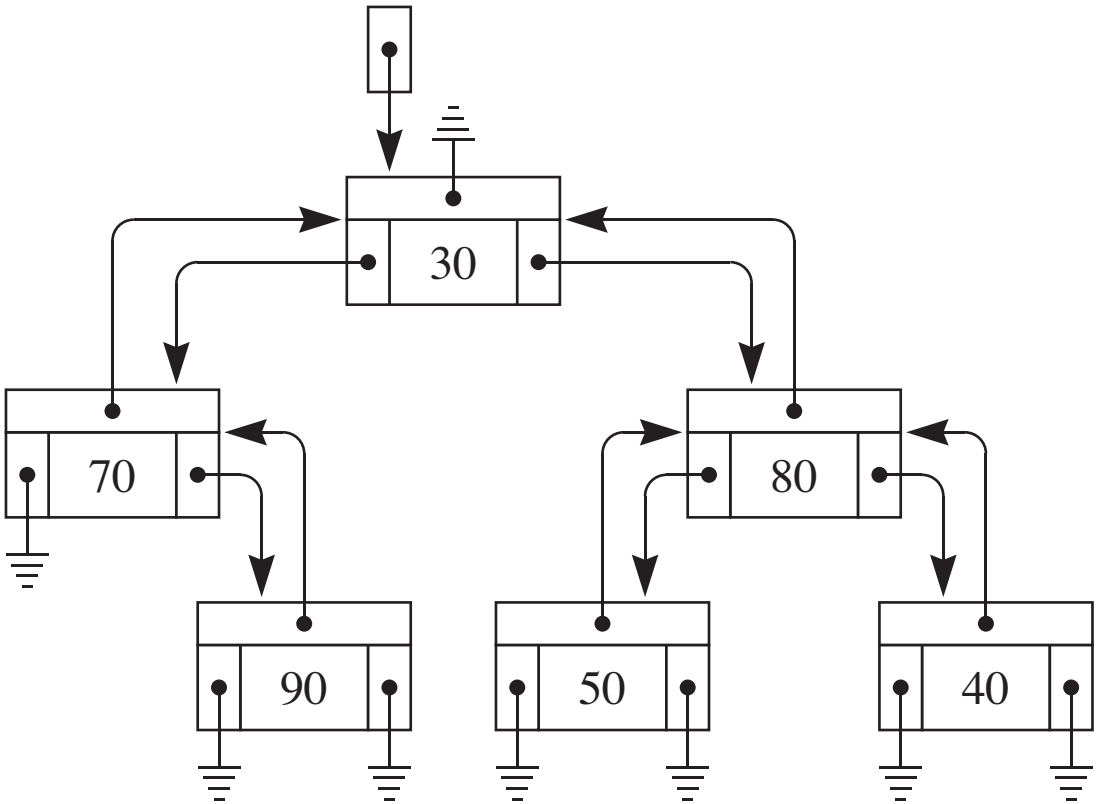
(b) A linked binary tree with left and right links.



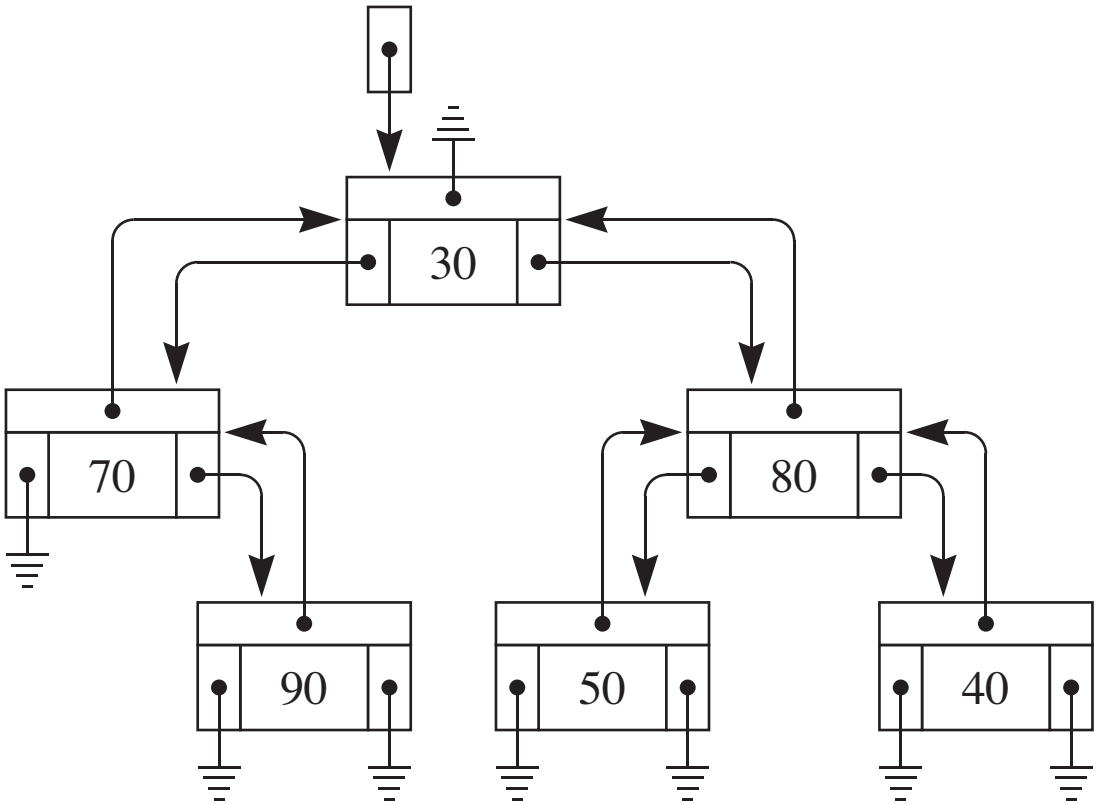
(b) A linked binary tree with left and right links.

```
template<class T> class BiTreeL {
private:
    shared_ptr<LNode<T>> _root;
```

```
template<class T> class LNode {
private:
    shared_ptr<LNode> _left;
    T _data;
    shared_ptr<LNode> _right;
```

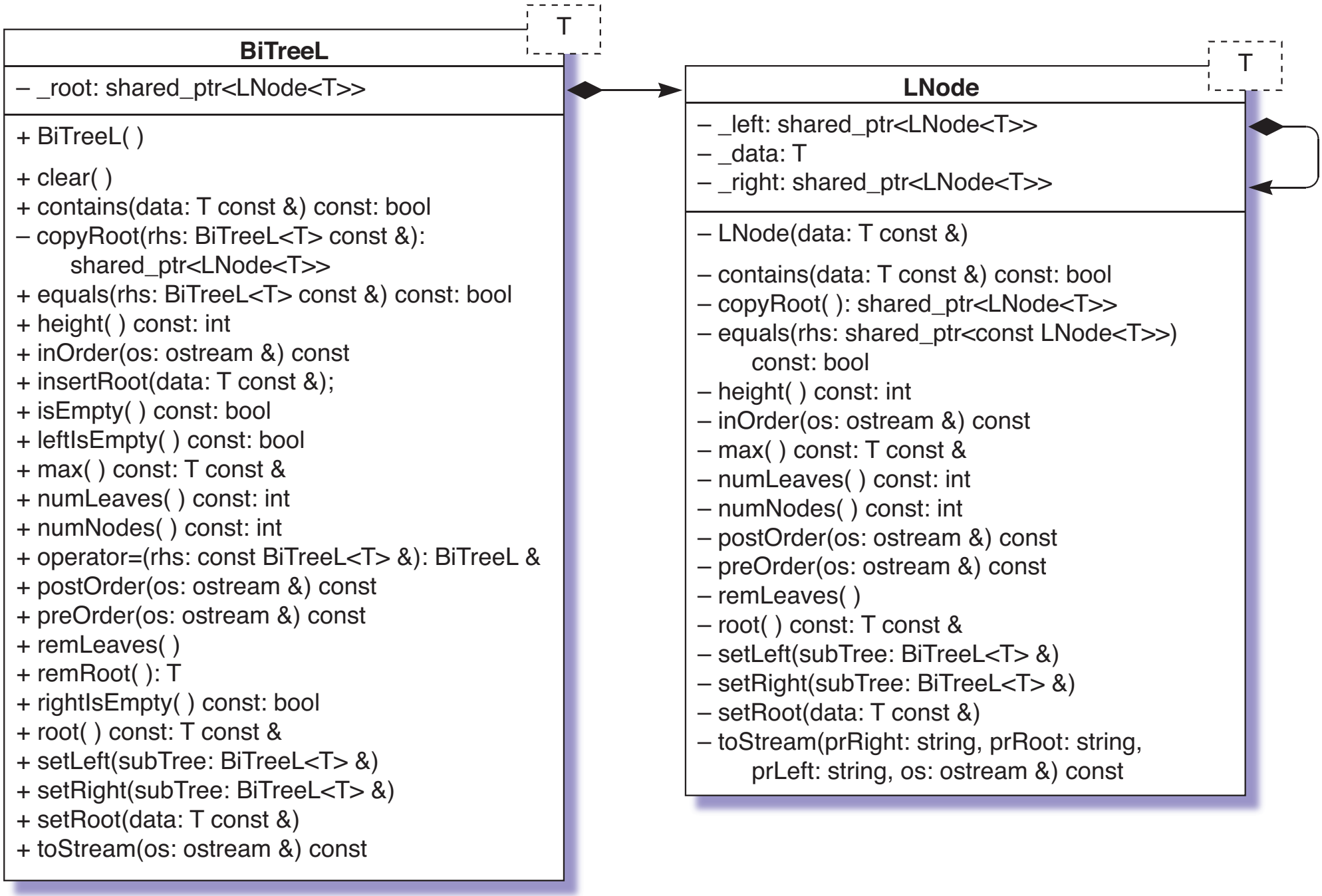


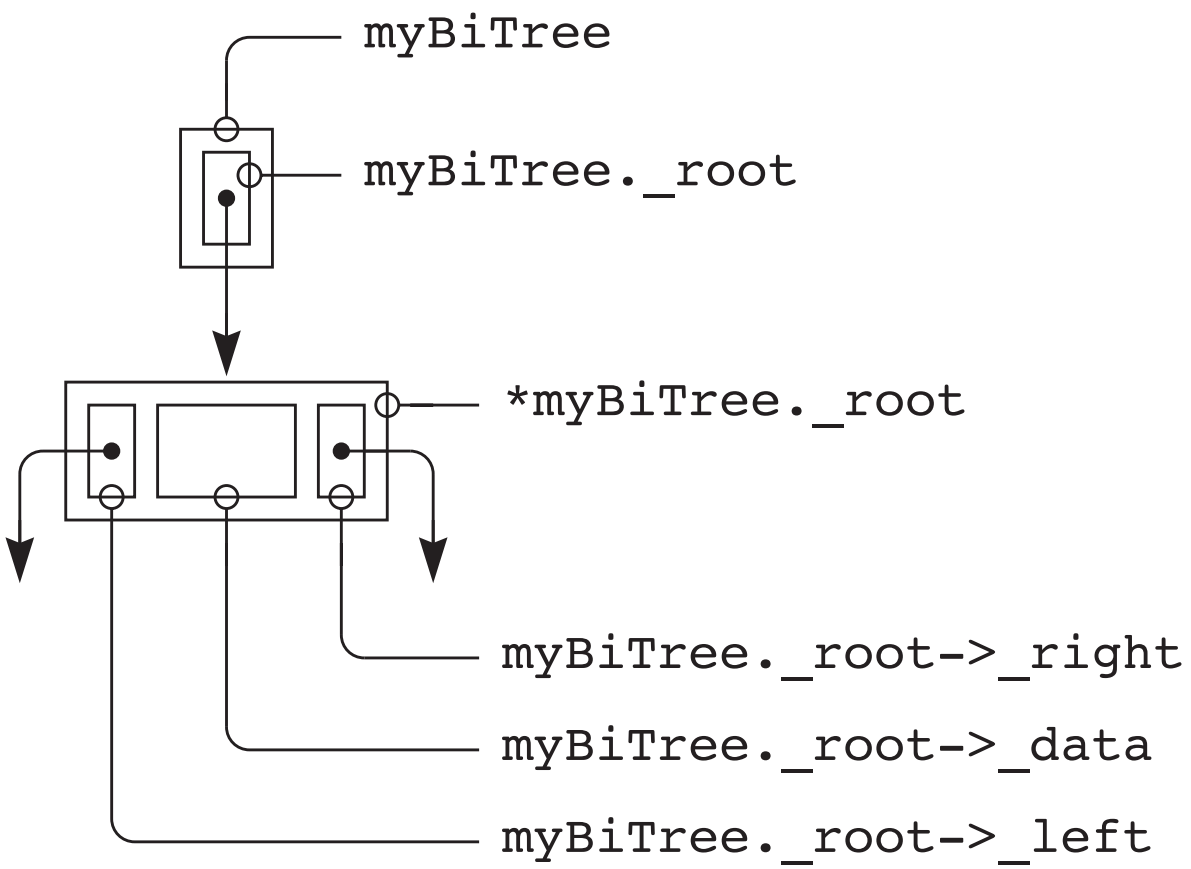
(c) A linked binary tree with left, right, and parent links.



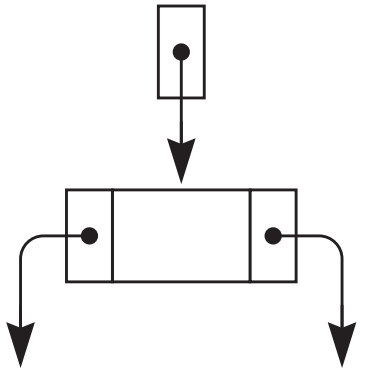
(c) A linked binary tree with left, right, and parent links.

```
template<class T> class BiTreeL {
private:
    shared_ptr<LNode<T>> _root;


  template<class T> class LNode {
  private:
      shared_ptr<LNode> _parent;
      shared_ptr<LNode> _left;
      T _data;
      shared_ptr<LNode> _right;
```


T
BiTreeL
– _root: shared_ptr<LNode<T>>
+ BiTreeL( )
+ clear( )
+ contains(data: T const &) const: bool
– copyRoot(rhs: BiTreeL<T> const &):
shared_ptr<LNode<T>>
+ equals(rhs: BiTreeL<T> const &) const: bool
+ height( ) const: int
+ inOrder(os: ostream &) const
+ insertRoot(data: T const &);
+ isEmpty( ) const: bool
+ leftIsEmpty( ) const: bool
+ max( ) const: T const &
+ numLeaves( ) const: int
+ numNodes( ) const: int
+ operator=(rhs: const BiTreeL<T> &): BiTreeL &
+ postOrder(os: ostream &) const
+ preOrder(os: ostream &) const
+ remLeaves( )
+ remRoot( ): T
+ rightIsEmpty( ) const: bool
+ root( ) const: T const &
+ setLeft(subTree: BiTreeL<T> &)
+ setRight(subTree: BiTreeL<T> &)
+ setRoot(data: T const &)
+ toStream(os: ostream &) const
T
LNode
– _left: shared_ptr<LNode<T>>
– _data: T
– _right: shared_ptr<LNode<T>>
– LNode(data: T const &)
– contains(data: T const &) const: bool
– copyRoot( ): shared_ptr<LNode<T>>
– equals(rhs: shared_ptr<const LNode<T>>)
const: bool
– height( ) const: int
– inOrder(os: ostream &) const
– max( ) const: T const &
– numLeaves( ) const: int
– numNodes( ) const: int
– postOrder(os: ostream &) const
– preOrder(os: ostream &) const
– remLeaves( )
– root( ) const: T const &
– setLeft(subTree: BiTreeL<T> &)
– setRight(subTree: BiTreeL<T> &)
– setRoot(data: T const &)
– toStream(prRight: string, prRoot: string,
prLeft: string, os: ostream &) const



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



(b) An abbreviated rendering.

Demo BiTreeL

Methods for output and characterization

```
void toStream(ostream &os) const;
// Post: A string representation of this tree is sent to os.

void preOrder(ostream &os) const;
// Post: A preorder representation of this tree is sent to os.

void inOrder(ostream &os) const;
// Post: An inorder representation of this tree is sent to os.

void postOrder(ostream &os) const;
// Post: A postorder representation of this tree is sent to os.
```

```
bool isEmpty() const;
// Post: true is returned if this tree is empty;
// otherwise, false is returned.

bool leftIsEmpty() const;
// Pre: This tree is not empty.
// Post: true is returned if the left subtree of this tree is empty;
// otherwise, false is returned.

bool rightIsEmpty() const;
// Pre: This tree is not empty.
// Post: true is returned if the right subtree of this tree is empty;
// otherwise, false is returned.
```

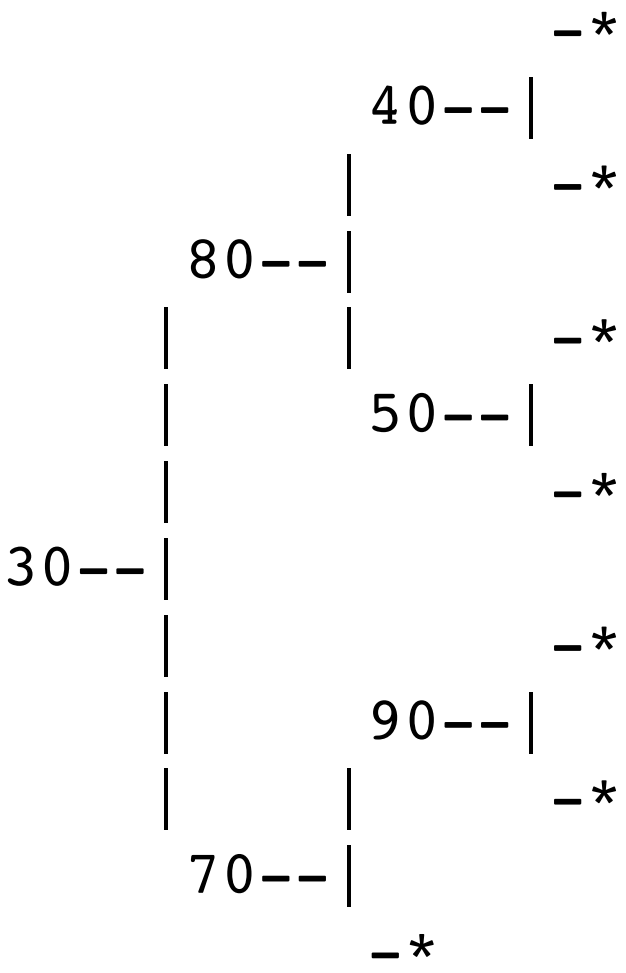
```
T const &root() const;
// Pre: This tree is not empty.
// Post: The root element of this tree is returned.

int numNodes() const;
// Post: The number of nodes of the host tree is returned.

int numLeaves() const;
// Post: The number of leaves of the host tree is returned.

int height() const;
// Post: The height of the host tree is returned.
```

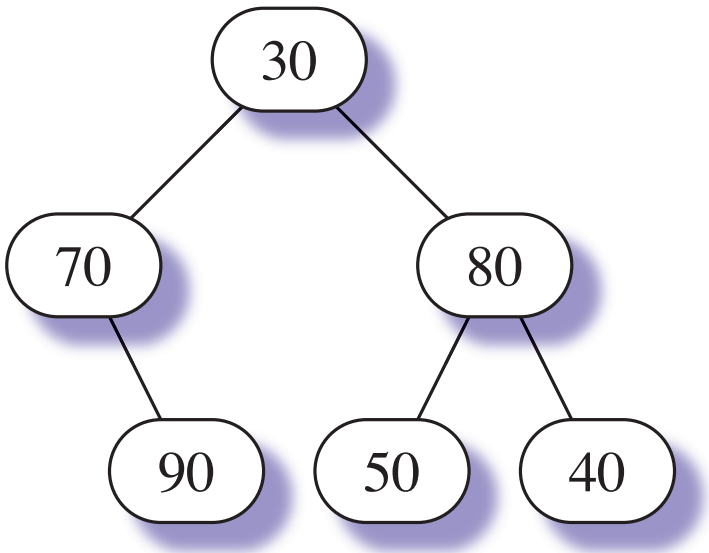
```
T const &max() const;
// Pre: This tree is not empty.
// Post: The maximum element of this tree is returned.

bool contains(T const &data) const;
// Post: true is returned if data is contained in this tree;
// otherwise, false is returned.

bool equals(BiTreeL<T> const &rhs) const;
// 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.
```

```
// ========= operator<< =========
template<class T>
ostream & operator<<(ostream &os, BiTreeL<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========
template<class T>
void BiTreeL<T>::toStream(ostream &os) const {
    if (_root == nullptr) {
        os << "*";
    }
    else {
        _root->toStream("", "", "", os);
    }
}
```

Node implementation of `toStream()` is not shown.

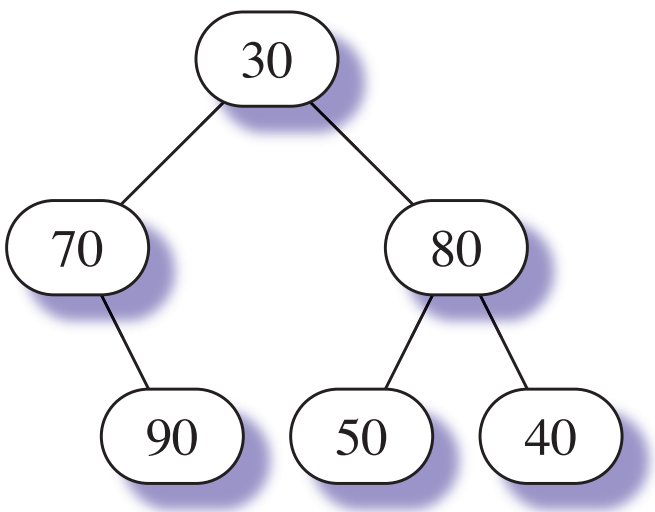
Figure 8.6





Preorder traversal:

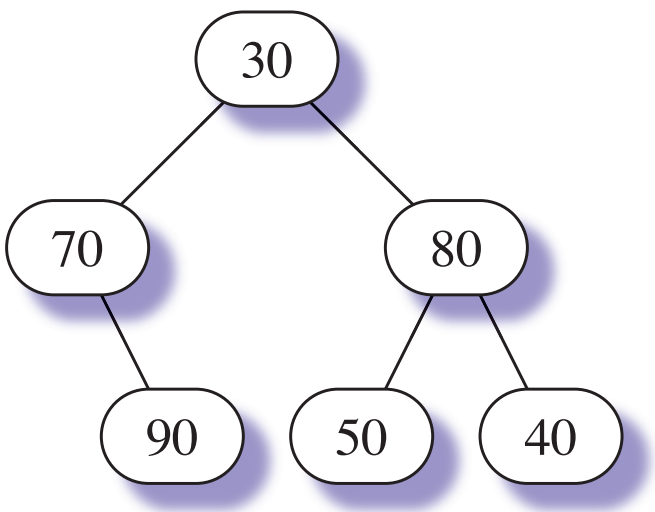
- Visit the root.
- Do a preorder traversal of the left subtree if it is not empty.
- Do a preorder traversal of the right subtree if it is not empty.



Preorder traversal:

Preorder traversal:

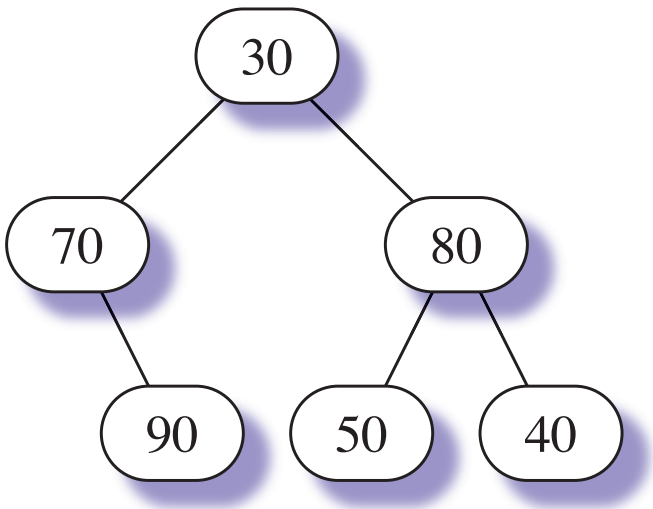
- Visit the root.
- Do a preorder traversal of the left subtree if it is not empty.
- Do a preorder traversal of the right subtree if it is not empty.



Preorder traversal: 30 70 90 80 50 40

Inorder traversal:

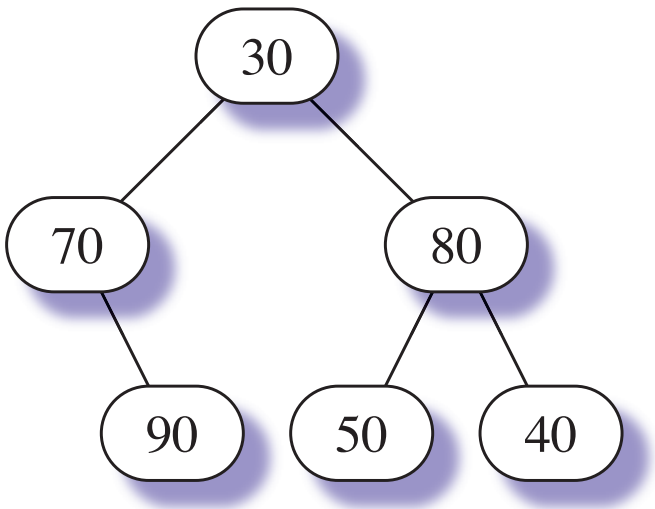
- Do an inorder traversal of the left subtree if it is not empty.
- Visit the root.
- Do an inorder traversal of the right subtree if it is not empty.



Inorder traversal:

Inorder traversal:

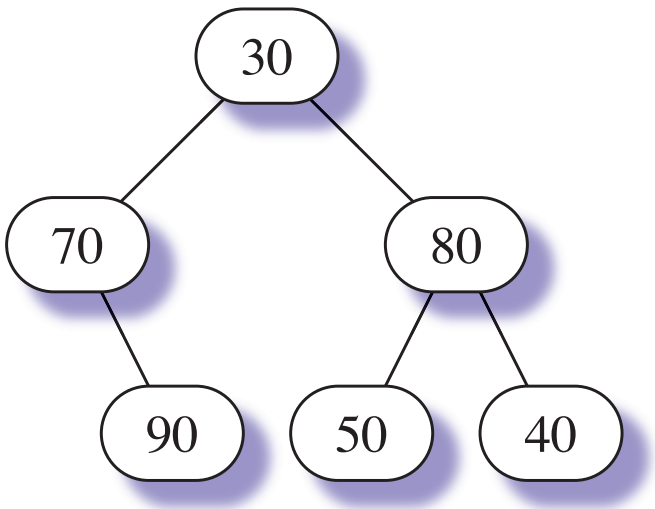
- Do an inorder traversal of the left subtree if it is not empty.
- Visit the root.
- Do an inorder traversal of the right subtree if it is not empty.



Inorder traversal: 70 90 30 50 80 40

Postorder traversal:

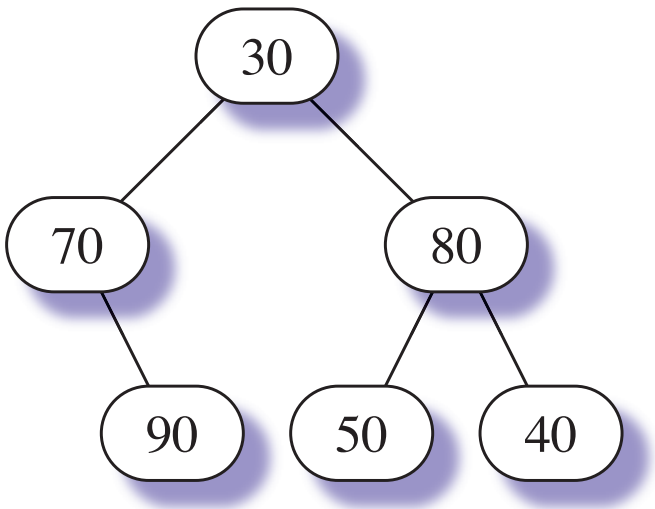
- Do a postorder traversal of the left subtree if it is not empty.
- Do a postorder traversal of the right subtree if it is not empty.
- Visit the root.



Postorder traversal:

Postorder traversal:

- Do a postorder traversal of the left subtree if it is not empty.
- Do a postorder traversal of the right subtree if it is not empty.
- Visit the root.



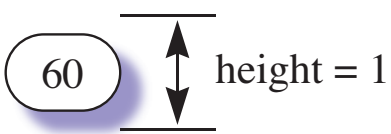
Postorder traversal: 90 70 50 40 80 30

```
// ========= preOrder =========
template<class T>
void BiTreeL<T>::preOrder(ostream &os) const {
    if (_root) {
        _root->preOrder(os);
    }
}

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

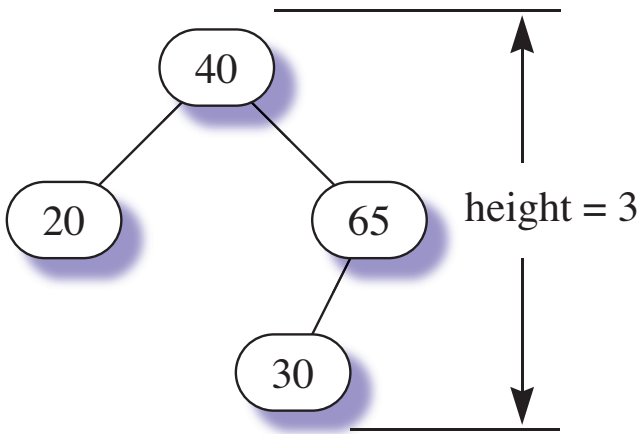
```
// ========== isEmpty ==========
template<class T>
bool BiTreeL<T>::isEmpty() const {
    return _root == nullptr;
}

// ========== leftIsEmpty ==========
template<class T>
bool BiTreeL<T>::leftIsEmpty() const {
    if (_root == nullptr) {
        cerr << "Precondition violated:"
             << "Cannot test left subtree of an empty tree."
             << endl;
        throw -1;
    }
    return _root->_left == nullptr;
}
```

```
// ========= root =========
template<class T>
T const &BiTreeL<T>::root() const {
    if (_root == nullptr) {
        cerr << "root precondition violated:"
             << "An empty tree has no root."
             << endl;
        throw -1;
    }
    return _root->root();
}

template<class T>
T const &LNode<T>::root() const {
    return _data;
}
```

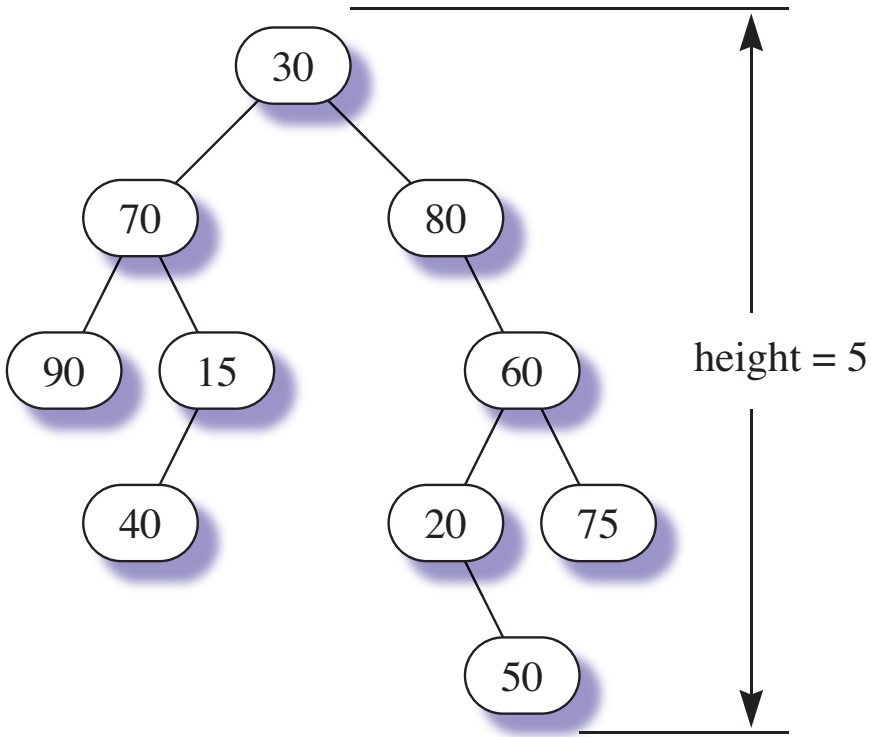
```
// ========= numNodes =========
template<class T>
int BiTreeL<T>::numNodes() const {
    return _root == nullptr ? 0 : _root->numNodes();
}

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



(a) A binary tree with leaf 60.



(b) A binary tree with leaves 20 and 30.



(c) A binary tree with leaves 90, 40, 50, and 75.

```
// ========= max =========
template<class T>
T const &BiTreeL<T>::max() const {
    if (_root == nullptr) {
        cerr << "Precondition violated: "
             << "An empty tree has no maximum."
             << endl;
        throw -1;
    }
    return _root->max();
}

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

Methods for construction and insertion

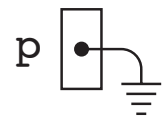
```
BiTreeL(BiTreeL<T> const &rhs) = delete;
// Copy constructor disabled. Private.

BiTreeL() = default;
// Post: This tree is initialized to be empty.

void insertRoot(T const &data);
// Pre: This tree is empty.
// Post: This tree has one root node containing data.

void setRoot(T const &data);
// Pre: This tree is not empty.
// Post: The root element of this tree is changed to data.
```

```
void setLeft(BiTreeL<T> &subTree);
// Pre: This tree is not empty.
// Post: The left child of this tree is subTree.
// subTree is the empty tree (cut setLeft, as opposed to copy setLeft).

void setRight(BiTreeL<T> &subTree);
// Pre: This tree is not empty.
// Post: The right child of this tree is subTree.
// subTree is the empty tree (cut setRight, as opposed to copy setRight).
```

```
BiTreeL & operator=(BiTreeL<T> const &rhs);
// Post: A deep copy of rhs is returned with garbage collection.

shared_ptr<LNode<T>> copyRoot(BiTreeL<T> const &rhs);
// Post: A deep copy of the root of rhs is returned.
```

```
// ========== Constructor ==========
template<class T>
LNode<T>::LNode(T const &data):
    _data(data) {
}
```

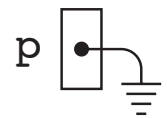
```
// ========= Constructor =========
template<class T>
LNode<T>::LNode(T const &data):
    _data(data) {
}
```



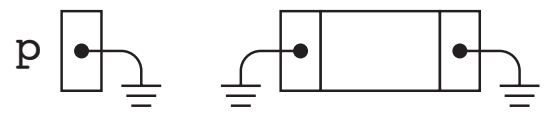
(a) Initial pointer.

```
p = shared_ptr<LNode<T>>(new LNode<T>(70))
```

```
// ========= Constructor =========
template<class T>
LNode<T>::LNode(T const &data):
    _data(data) {
}
```



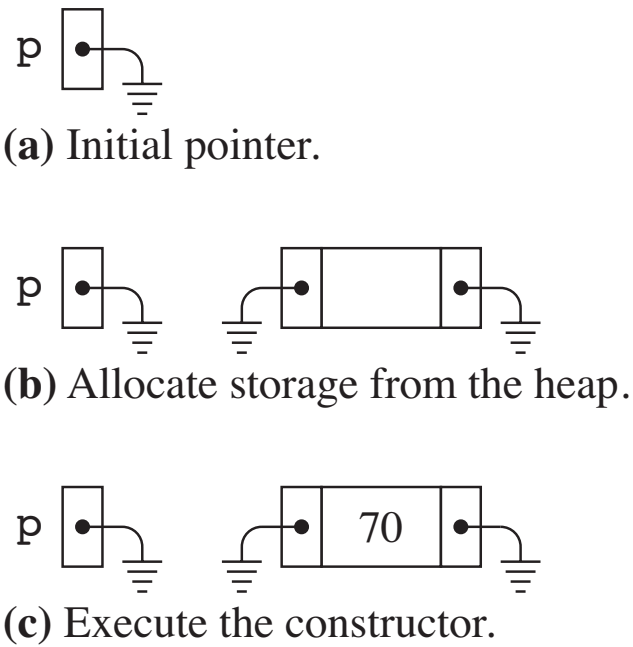
(a) Initial pointer.



(b) Allocate storage from the heap.

```
p = shared_ptr<LNode<T>>(new LNode<T>(70))
```

```
// ========= Constructor =========
template<class T>
LNode<T>::LNode(T const &data):
    _data(data) {
}
```



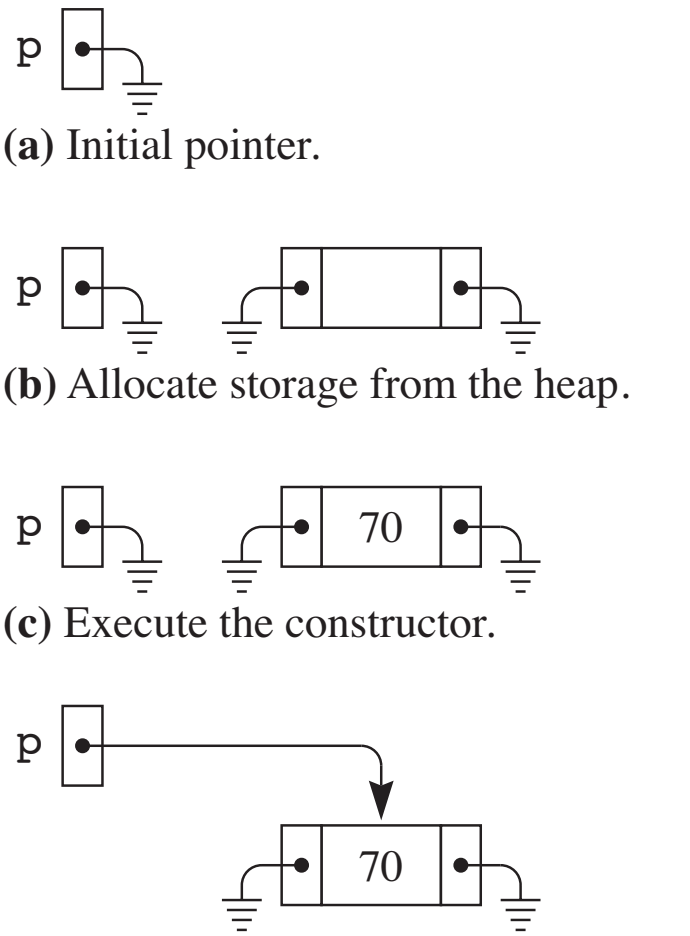
(a) Initial pointer.

(b) Allocate storage from the heap.

(c) Execute the constructor.

```
p = shared_ptr<LNode<T>>(new LNode<T>(70))
```

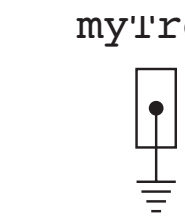
```
// ========= Constructor =========
template<class T>
LNode<T>::LNode(T const &data):
    _data(data) {
}
```




```
p = shared_ptr<LNode<T>>(new LNode<T>(70))
```

```
// ========== insertRoot ==========
template<class T>
void BiTreeL<T>::insertRoot(T const &data) {
    if (_root != nullptr) {
        cerr << "insertRoot precondition violated:"
             << "Cannot insert root into a non empty tree"
             << endl;
        throw -1;
    }
    _root = shared_ptr<LNode<T>>(new LNode<T>(data));
}
```

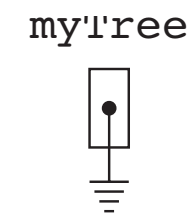
```
// ========== insertRoot ==========
template<class T>
void BiTreeL<T>::insertRoot(T const &data) {
    if (_root != nullptr) {
        cerr << "insertRoot precondition violated:"
             << "Cannot insert root into a non empty tree"
             << endl;
        throw -1;
    }
    _root = shared_ptr<LNode<T>>(new LNode<T>(data));
}
```



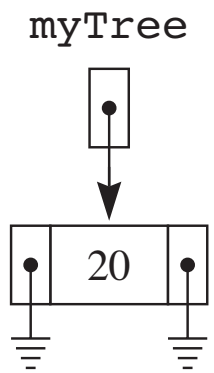
(a) Original tree.

```
myTree.insertRoot(20)
```

```
// ========== insertRoot ==========
template<class T>
void BiTreeL<T>::insertRoot(T const &data) {
    if (_root != nullptr) {
        cerr << "insertRoot precondition violated:"
             << "Cannot insert root into a non empty tree"
             << endl;
        throw -1;
    }
    _root = shared_ptr<LNode<T>>(new LNode<T>(data));
}
```

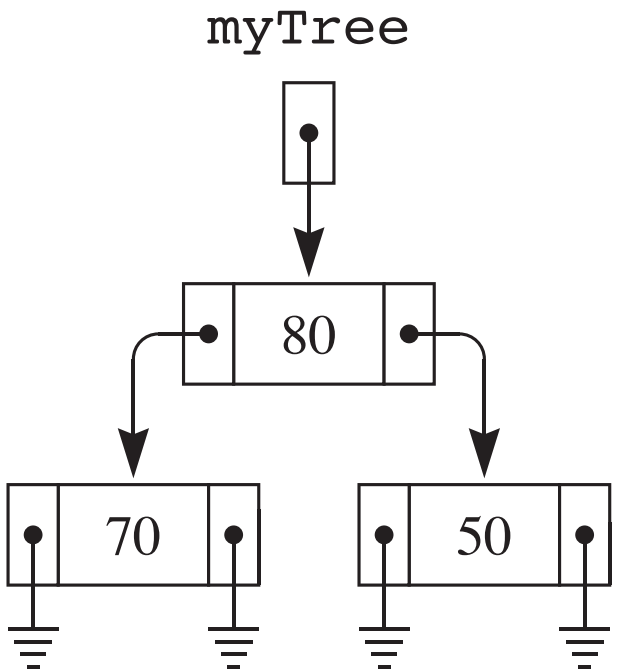


(a) Original tree.



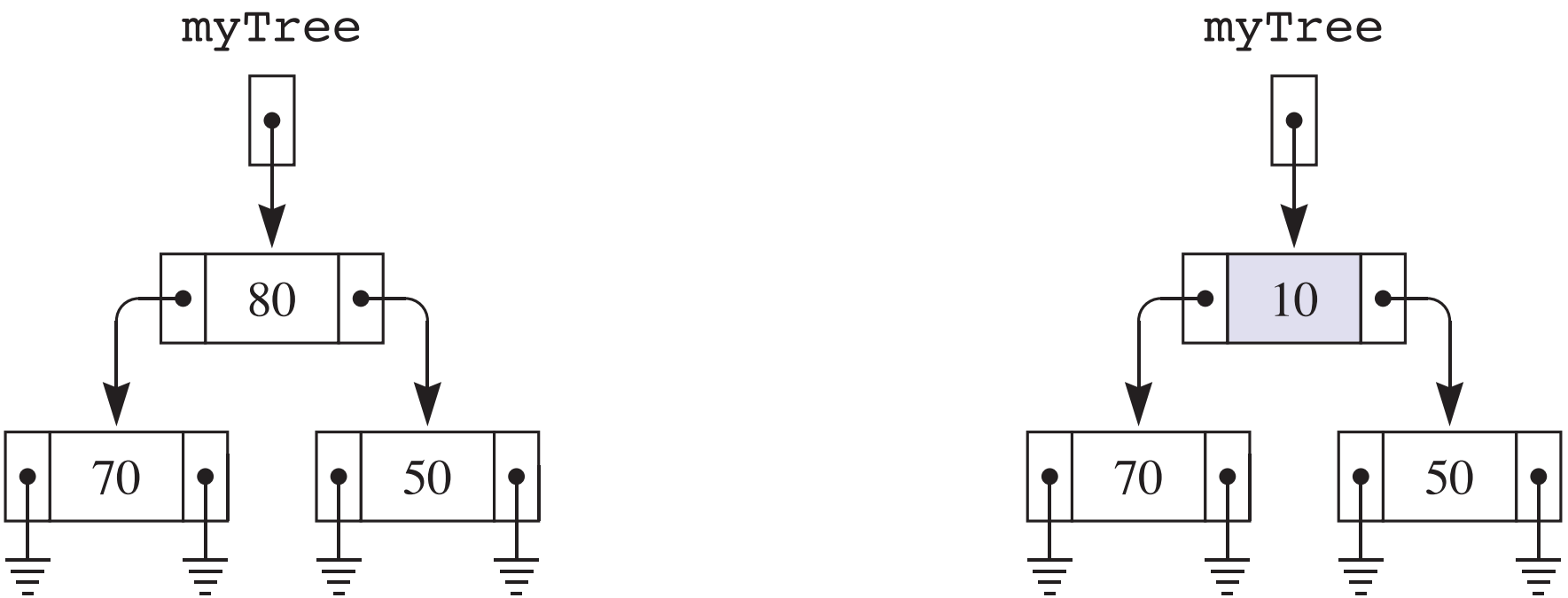
(b) After inserting root.

```
myTree.insertRoot(20)
```



(a) Original tree.

```
myTree.setRoot(10)
```

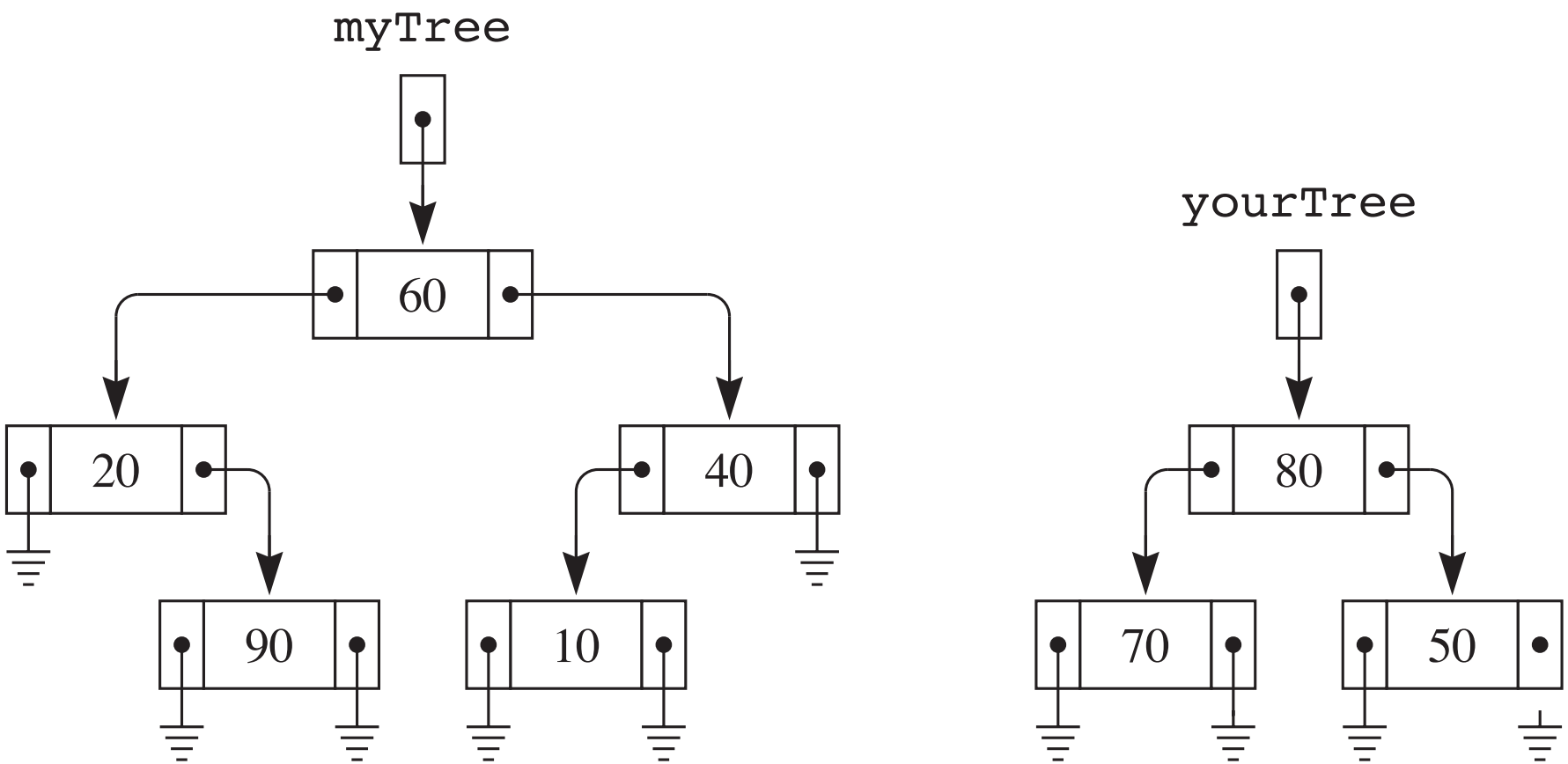


(a) Original tree.

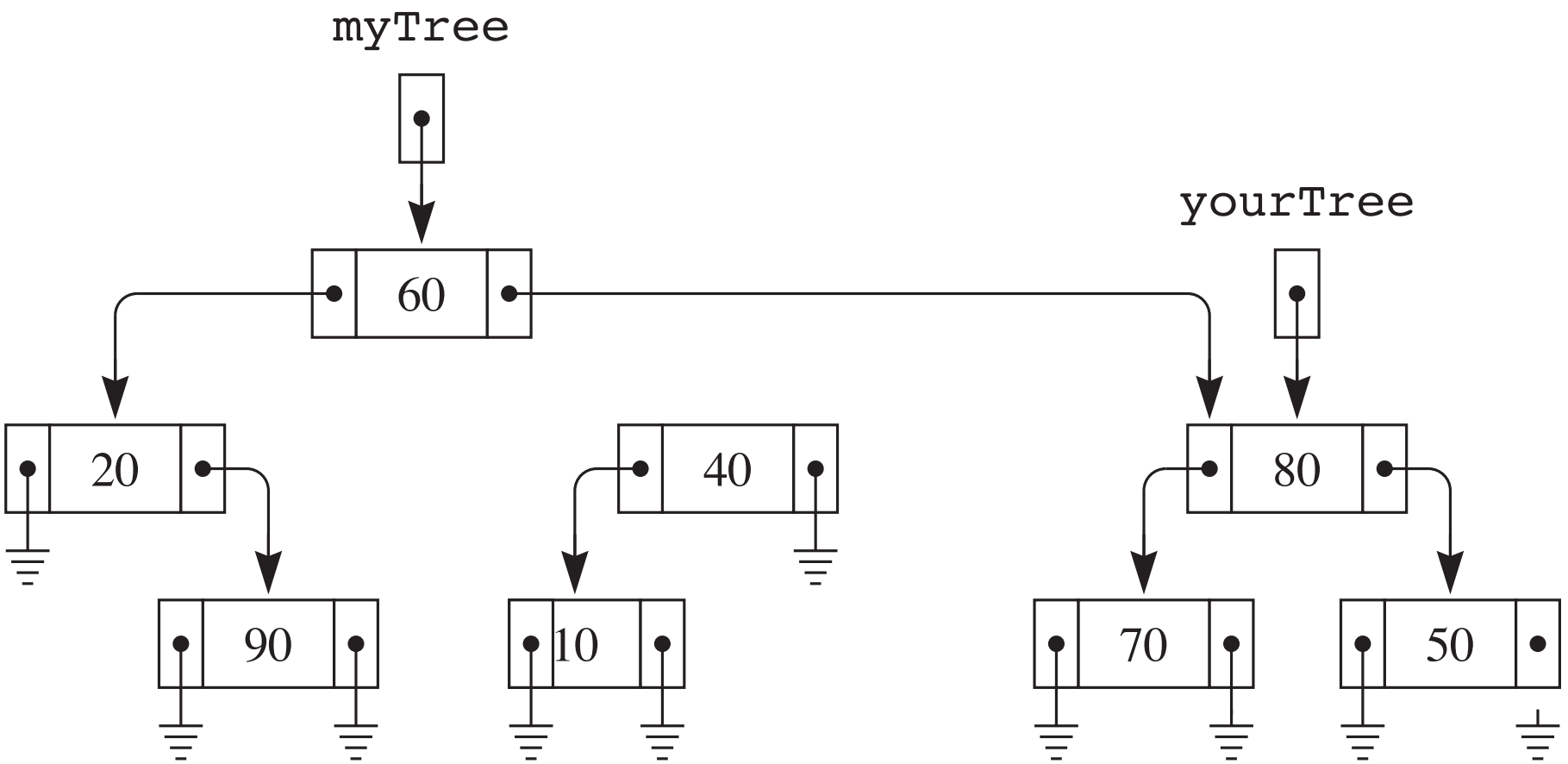
(b) After setting root.

```
myTree.setRoot(10)
```

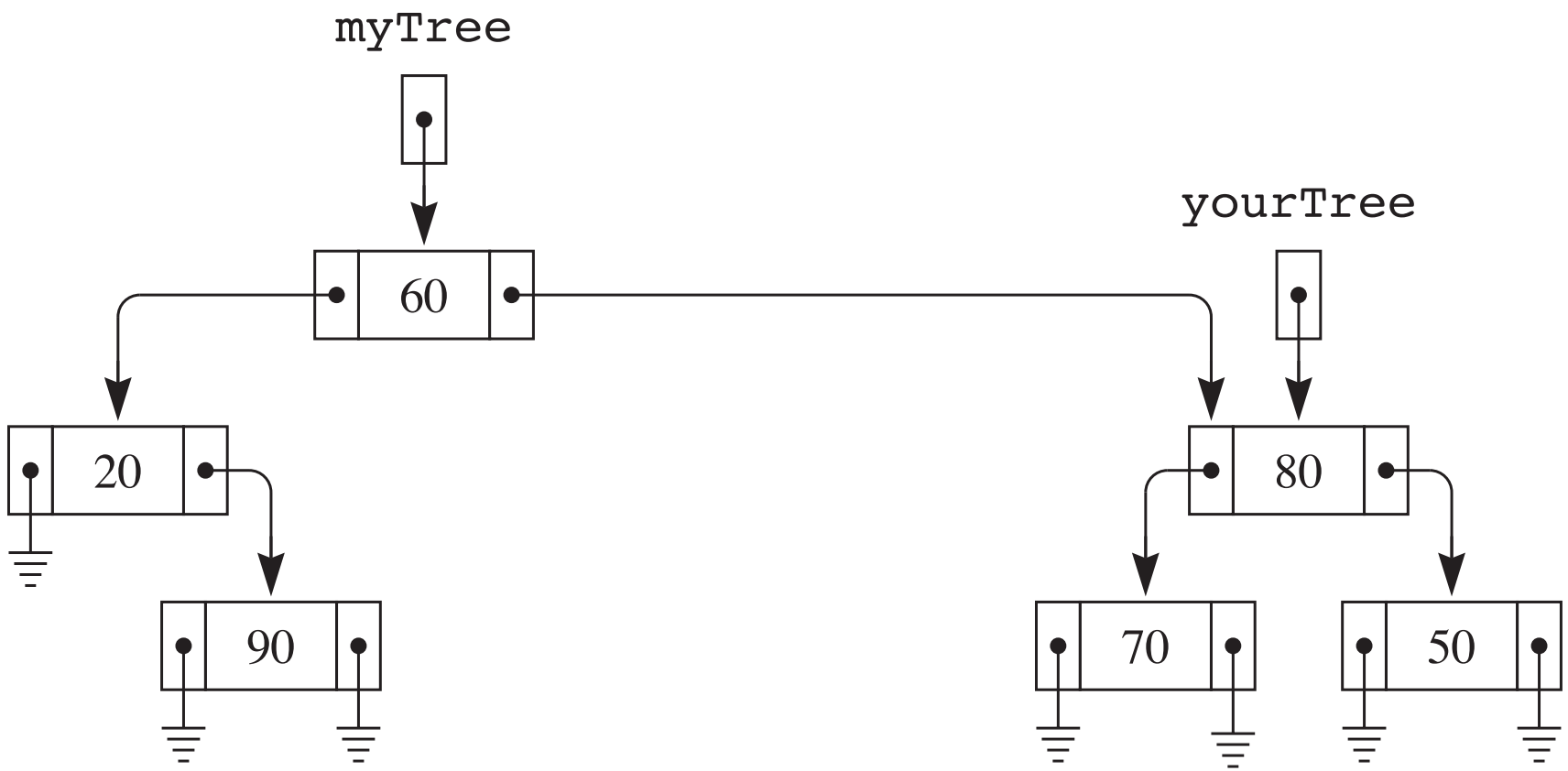
```
// ========= setRight =========
template<class T>
void BiTreeL<T>::setRight(BiTreeL<T> &subTree) {
    if (_root == nullptr) {
        cerr << "Precondition violated:"
             << "Cannot set right on an empty tree."
             << endl;
        throw -1;
    }
    _root->setRight(subTree);
}

template<class T>
void LNode<T>::setRight(BiTreeL<T> &subTree) {
    _right = subTree._root;
    subTree._root.reset();
}
```



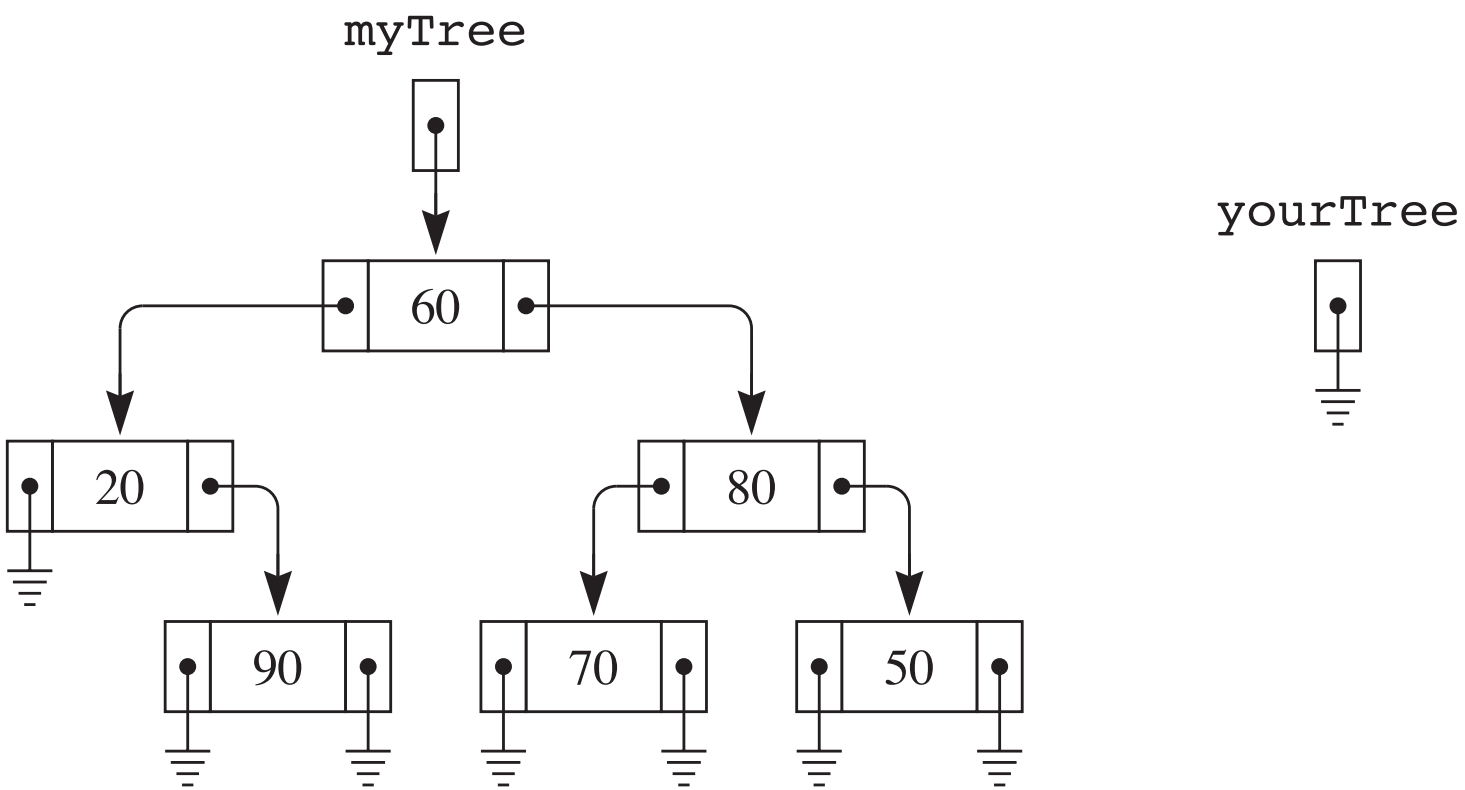

```
myTree.setRight(yourTree)
```




```
_right = subtree._root
```



Automatic garbage collection




```
subtree._root.reset()
```

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

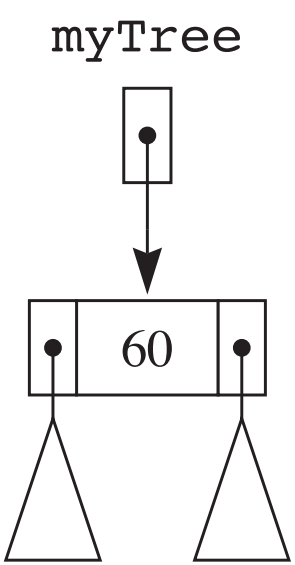
```
// ========= copyRoot =========
template<class T>
shared_ptr<LNode<T>> BiTreeL<T>::copyRoot(BiTreeL<T> const &rhs) {
    return rhs.isEmpty() ? nullptr : rhs._root->copyRoot();
}

template<class T>
shared_ptr<LNode<T>> LNode<T>::copyRoot() {
    shared_ptr<LNode<T>> result =
        shared_ptr<LNode<T>>(new LNode<T >(_data));
    if (_left) {
        result->_left = _left->copyRoot();
    }
    if (_right) {
        result->_right = _right->copyRoot();
    }
    return result;
}
```

Methods for removal

```
void clear();
// Post: This tree is cleared to the empty tree.

T remRoot();
// Pre: This tree is not empty and its root
// has at least one empty child.
// Post: The root node is removed from this tree
// and its element is returned.

void remLeaves();
// Post: The leaves are removed from this tree.
```

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

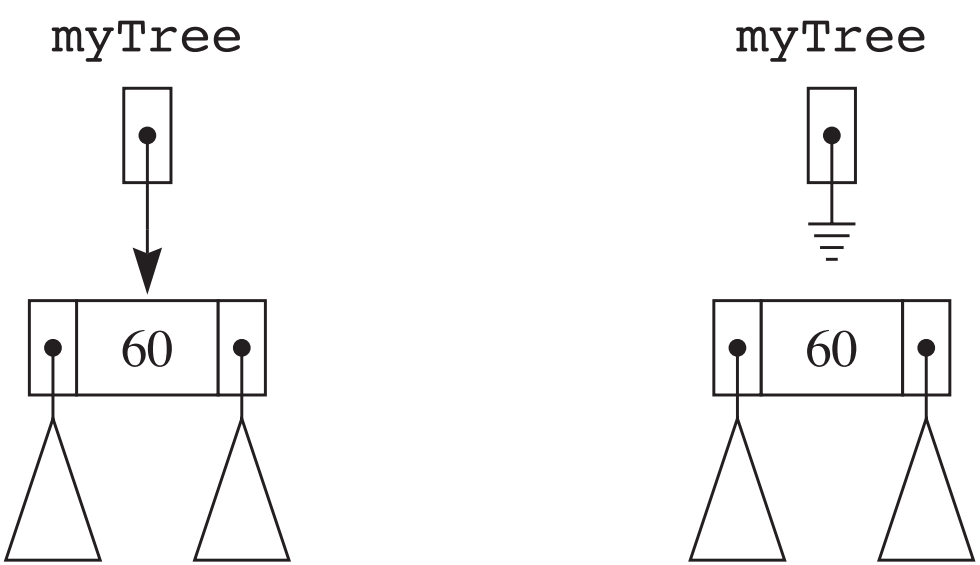
```
// ========= clear =========
template<class T>
void BiTreeL<T>::clear() {
    cerr << "BiTreeL<T>::clear: "
         << Exercise for the student." << endl;
    throw -1;
}
```



(a) Original.

myTree.clear()

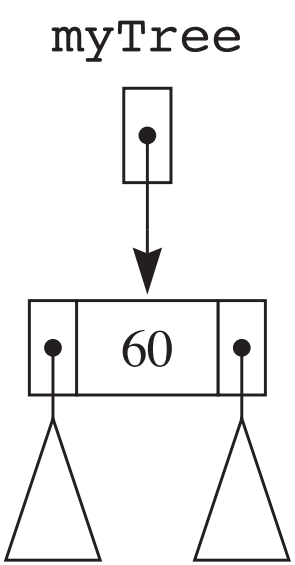
```
// ========= clear =========
template<class T>
void BiTreeL<T>::clear() {
    cerr << "BiTreeL<T>::clear: "
         << Exercise for the student." << endl;
    throw -1;
}
```



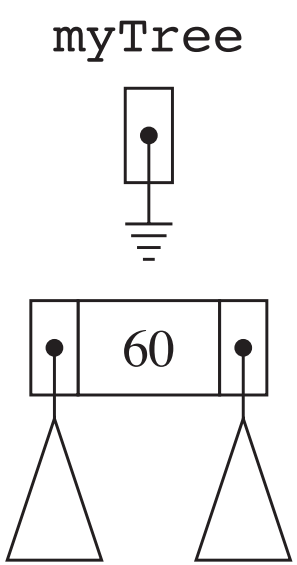
(a) Original. **(b)** Set to nil.

`myTree.clear()`

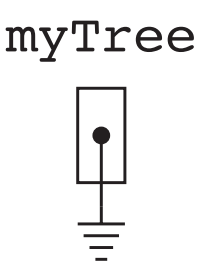
```
// ========= clear =========
template<class T>
void BiTreeL<T>::clear() {
    cerr << "BiTreeL<T>::clear: "
         << Exercise for the student." << endl;
    throw -1;
}
```



(a) Original.

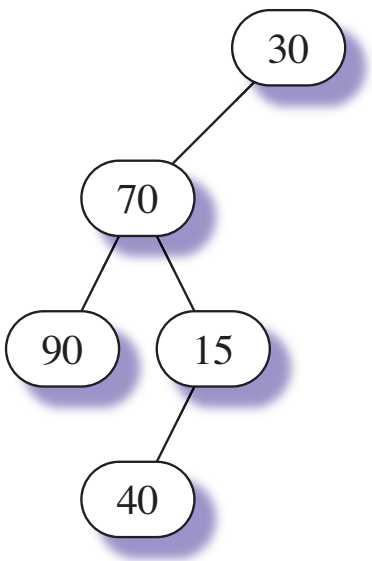


(b) Set to nil.



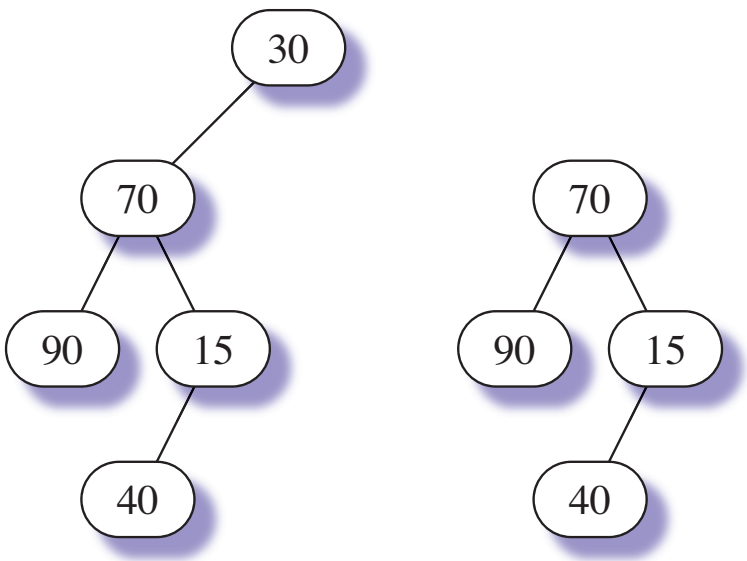
(c) Automatic garbage collection.

myTree.clear()



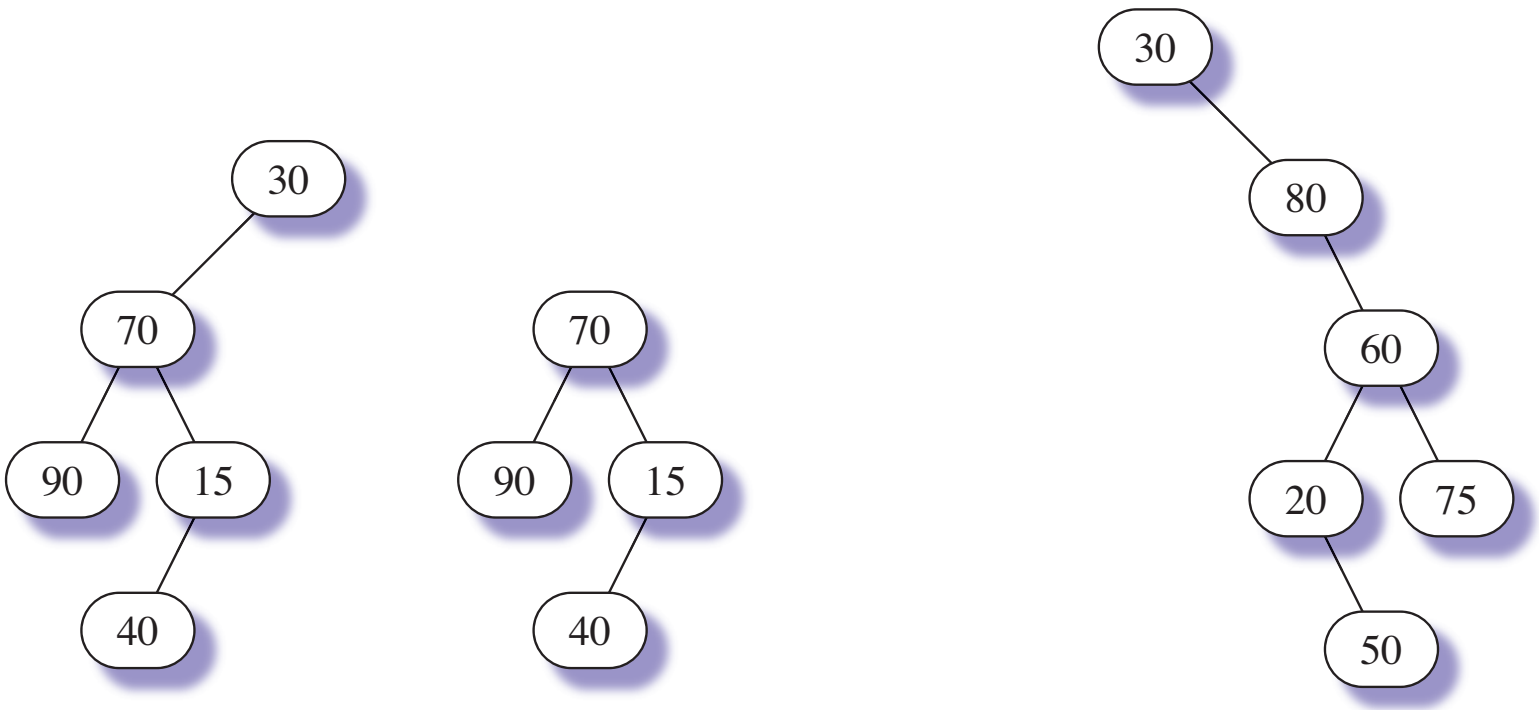
(a) Original tree.

`remRoot()`



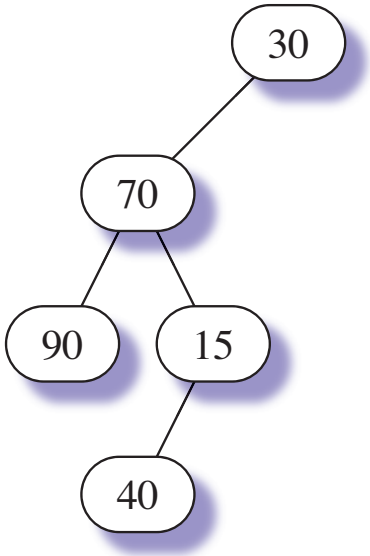
(a) Original tree. **(b)** After `remRoot()`.

`remRoot()`

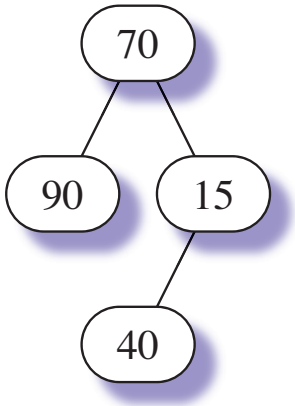


(a) Original tree. **(b)** After `remRoot()`. **(c)** Original tree.

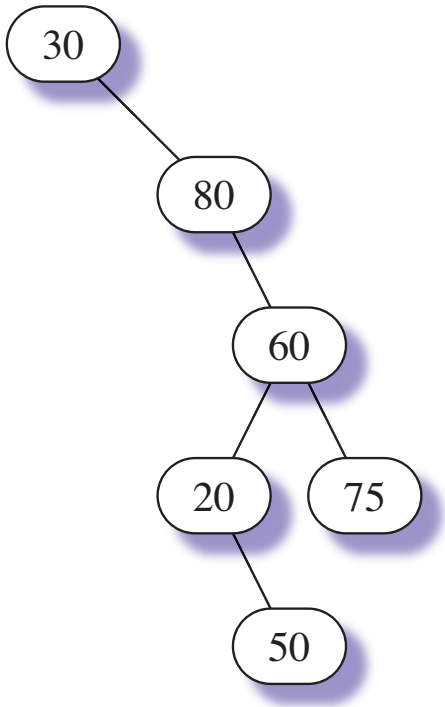
`remRoot()`



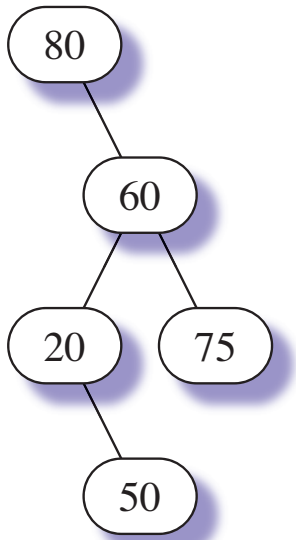
(a) Original tree.



(b) After `remRoot()`.

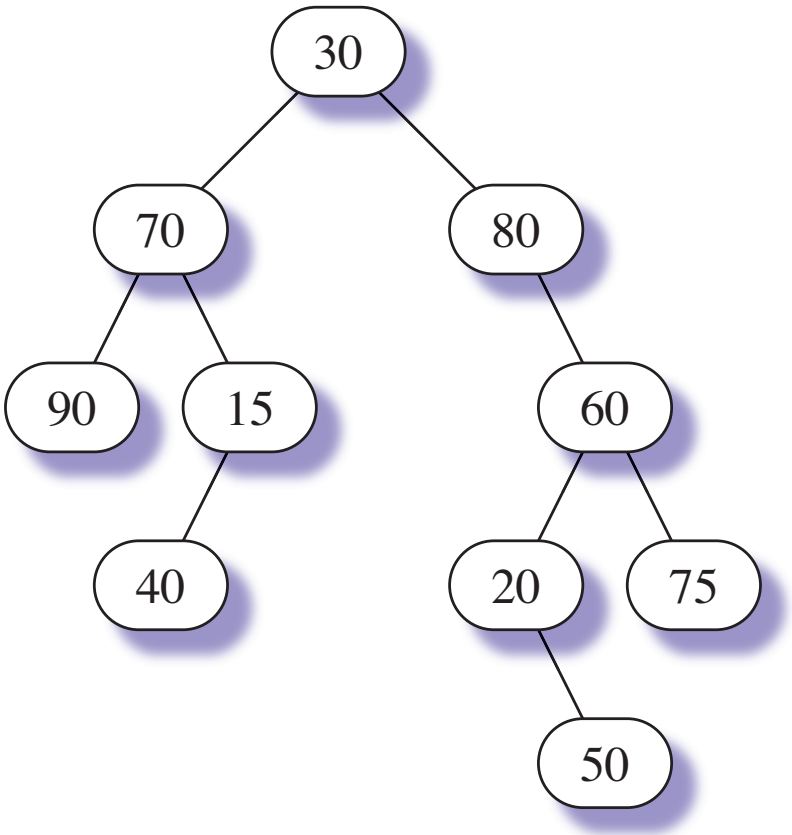


(c) Original tree.



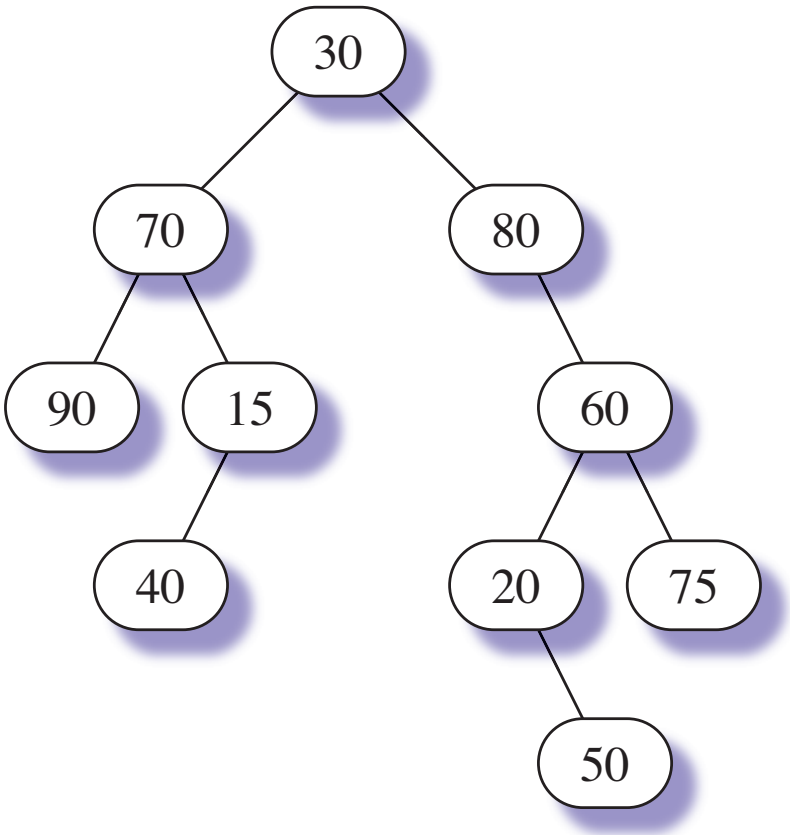
(d) After `remRoot()`.

`remRoot()`

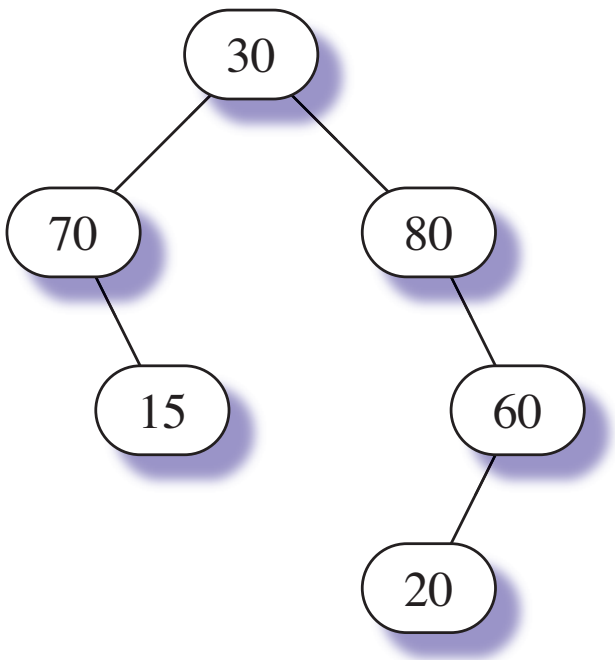


(a) Original tree.

`remLeaves()`



(a) Original tree.



(b) After `remLeaves()`.

`remLeaves()`