```
//7.1
template <typename E>                // base element type
  class Position<E> {                // a node position
  public:
    E& operator*();                  // get element
    Position parent() const;         // get parent
    PositionList children() const;   // get node's children
    bool isRoot() const;             // root node?
    bool isExternal() const;         // external node?
  };
//7.2
template <typename E>                // base element type
  class Tree<E> {
  public:                            // public types
    class Position;              // a node position
    class PositionList;              // a list of positions
  public:                            // public functions
    int size() const;                // number of nodes
    bool empty() const;              // is tree empty?
    Position root() const;           // get the root
    PositionList positions() const;        // get positions of all nodes
  };

//7.4
int depth(const Tree& T, const Position& p) {
    if (p.isRoot())
      return 0;                      // root has depth 0
    else
      return 1 + depth(T, p.parent());// 1 + (depth of parent)
  }
//7.6
int height2(const Tree& T, const Position& p) {
    if (p.isExternal()) return 0;// leaf has height 0
    int h = 0;
    PositionList ch = p.children();    // list of children
    for (Iterator q = ch.begin(); q != ch.end(); ++q)
      h = max(h, height2(T, *q));
    return 1 + h;  // 1 + max height of children
```

```
  }
//7.15
template <typename E>                          // base element type
  class Position<E> {                          // a node position
  public:
    E& operator*();                            // get element
    Position left() const;                     // get left child
    Position right() const;                    // get right child
    Position parent() const;                   // get parent
    bool isRoot() const;                       // root of tree?
    bool isExternal() const;                   // an external node?
};
//7.16
template <typename E>                      // base element type
  class BinaryTree<E> {                    // binary tree
  public:                                  // public types
    class Position;                  // a node position
    class PositionList;                    // a list of positions
  public:                                  // member functions
    int size() const;                      // number of nodes
    bool empty() const;                    // is tree empty?
    Position root() const;                 // get the root
    PositionList positions() const;        // list of nodes
  };

//7.17
struct Node {                                  // a node of the tree
      Elem    elt;                                    // element value
      Node*   par;                                    // parent
      Node*   left;                                   // left child
      Node*   right;                                  // right child
      Node() : elt(), par(NULL), left(NULL), right(NULL) { } // constructor
    };

//7.18
class Position {                               // position in the tree
    private:
      Node* v;                                        // pointer to the node
```

```
    public:
      Position(Node* _v = NULL) : v(_v) { }          // constructor
      Elem& operator*()                              // get element
        { return v->elt; }
      Position left() const                          // get left child
        { return Position(v->left); }
      Position right() const                         // get right child
        { return Position(v->right); }
      Position parent() const                        // get parent
        { return Position(v->par); }
      bool isRoot() const                            // root of the tree?
        { return v->par == NULL; }
      bool isExternal() const                        // an external node?
        { return v->left == NULL && v->right == NULL; }
      friend class LinkedBinaryTree;                 // give tree access
    };
    typedef std::list<Position> PositionList;        // list of positions
```

//7.19

```
typedef int Elem;                                    // base element type
  class LinkedBinaryTree {
  protected:
    // insert Node declaration here...
  public:
    // insert Position declaration here...
  public:
    LinkedBinaryTree();                              // constructor
    int size() const;                                // number of nodes
    bool empty() const;                              // is tree empty?
    Position root() const;                           // get the root
    PositionList positions() const;                  // list of nodes
    void addRoot();                                  // add root to empty
tree
    void expandExternal(const Position& p);          // expand external
node
    Position removeAboveExternal(const Position& p); // remove p and parent
    // housekeeping functions omitted...
  protected:                                         // local utilities
```

```
    void preorder(Node* v, PositionList& pl) const;     // preorder utility
  private:
    Node* _root;                                        // pointer to the root
    int n;                                              // number of nodes
  };
//7.20
LinkedBinaryTree::LinkedBinaryTree()                    // constructor
    : _root(NULL), n(0) { }
  int LinkedBinaryTree::size() const                    // number of nodes
    { return n; }
  bool LinkedBinaryTree::empty() const                  // is tree empty?
    { return size() == 0; }
  LinkedBinaryTree::Position LinkedBinaryTree::root() const // get the root
    { return Position(_root); }
  void LinkedBinaryTree::addRoot()                      // add root to empty
tree
    { _root = new Node; n = 1; }

//7.21
void LinkedBinaryTree::expandExternal(const Position& p) {
    Node* v = p.v;                                      // p's node
    v->left = new Node;                                         // add a new
left child
    v->left->par = v;                                   // v is its parent
    v->right = new Node;                                // and a new right child
    v->right->par = v;                                  // v is its parent
    n += 2;                                             // two more nodes
  }

//7.22
  LinkedBinaryTree::Position                            // remove p and parent
  LinkedBinaryTree::removeAboveExternal(const Position& p) {
    Node* w = p.v;  Node* v = w->par;                           // get p's node
and parent
    Node* sib = (w == v->left ?  v->right : v->left);
    if (v == _root) {                                   // child of root?
      _root = sib;                                      // ...make sibling root
      sib->par = NULL;
```

```
    }
    else {
      Node* gpar = v->par;                          // w's grandparent
      if (v == gpar->left) gpar->left = sib;        // replace parent by
sib
      else gpar->right = sib;
      sib->par = gpar;
    }
    delete w; delete v;                             // delete removed
nodes
    n -= 2;                                         // two fewer nodes
    return Position(sib);
  }

//7.23
LinkedBinaryTree::PositionList LinkedBinaryTree::positions() const {
  PositionList pl;
  preorder(_root, pl);                              // preorder traversal
  return PositionList(pl);                          // return resulting list
}
                                                    // preorder traversal
void LinkedBinaryTree::preorder(Node* v, PositionList& pl) const {
  pl.push_back(Position(v));                        // add this node
  if (v->left != NULL)                              // traverse left
subtree
    preorder(v->left, pl);
  if (v->right != NULL)                             // traverse right
subtree
    preorder(v->right, pl);
}
```