```
// ******************************************************
// Header file BT.h for the ADT binary tree.
// ******************************************************

typedef desired-type-of-tree-item treeItemType;

typedef void (*functionType)(treeItemType& AnItem);

struct treeNode;  // defined in implementation file
typedef treeNode* ptrType;  // pointer to node

class binTreeClass
{
  public:
    // constructors and destructor:

      // Create an empty binary tree - NULL root ptr
    binTreeClass();

      /* Create a binary tree with root node containing RootItem
     and NULL left and right child pointers */
    binTreeClass(const treeItemType& RootItem);

      /* Create a binary tree BT with root node containing
     RootItem, and make the left and right subtrees of BT
     equal to copies of LeftTree and RightTree. */
    binTreeClass(const treeItemType& RootItem,
                 const binTreeClass& LeftTree,
                 const binTreeClass& RightTree);

      /* Create a binary tree BT equal to a copy of 'Tree'. */
    binTreeClass(const binTreeClass& Tree);

      /* The definition of this constructor appears further below
     - in the 'protected' section. */
      //   binTreeClass(ptrType NodePtr);  // constructor

      /* Remove and deallocate all the nodes, one-by-one.*/
    virtual ~binTreeClass();

      // binary tree operations:

    virtual bool BinaryTreeIsEmpty() const;

      /* If the tree is not empty, return a copy of the item in
     the root node. */
```

```
virtual treeItemType RootData() const;

  /* Check to see if the tree has a root node.  If not,
 create one.  Next, store a copy of 'NewItem' into the
 root node. */
virtual void SetRootData(const treeItemType& NewItem);

  /* If the tree has a root with no left child, then give it
 a left child with 'NewItem' stored there, and give the
 left child NULL subtrees. */
virtual void AttachLeft(const treeItemType& NewItem,
                        bool& Success);

  /* If the tree has a root with no right child, then give it
 a right child with 'NewItem' stored there, and give the
 right child NULL subtrees. */
virtual void AttachRight(const treeItemType& NewItem,
                         bool& Success);

  /* If the tree has a root with no left child, then give it
 a left subtree equal to a copy of 'LeftTree'. */
virtual void AttachLeftSubtree(const binTreeClass& LeftTree,
                               bool& Success);

  /* If the tree has a root with no right child, then give it
 a right subtree equal to a copy of 'RightTree'. */
virtual void AttachRightSubtree(const binTreeClass& RightTree,
                                bool& Success);

  /* If the tree is not empty, detach its left subtree, make
 it into an object of binTreeClass, and return that
 object as 'LeftTree'. */
virtual void DetachLeftSubtree(binTreeClass& LeftTree,
                               bool& Success);

  /* If the tree is not empty, detach its right subtree, make
 it into an object of binTreeClass, and return that
 object as 'RightTree'.*/
virtual void DetachRightSubtree(binTreeClass& RightTree,
                                bool& Success);

  /* If this tree is not empty, return a binary tree object
 that is a copy of the left subtree of this tree. (If
 this tree is empty, return an empty tree.) */
virtual binTreeClass LeftSubtree() const;
```

```
    /* If this tree is not empty, return a binary tree object
     that is a copy of the right subtree of this tree. (If
     this tree is empty, return an empty tree.) */
    virtual binTreeClass RightSubtree() const;

    /* Traverse the tree in pre-order and apply 'Visit' to each
     node. */
    virtual void PreorderTraverse(functionType Visit);

    /* Traverse the tree in in-order and apply 'Visit' to each
     node. */
    virtual void InorderTraverse(functionType Visit);

    /* Traverse the tree in post-order and apply 'Visit' to
     each node. */
    virtual void PostorderTraverse(functionType Visit);

    // overloaded operator:
    /* Make this tree equal to a copy of Rhs. */
    virtual binTreeClass& operator=(const binTreeClass& Rhs);

  protected:

    /* Create a binary tree BT by making 'NodePtr' the root
     pointer.  This operation does no memory allocation for
     tree nodes- just makes the root pointer of BT point to
     whatever memory 'NodePtr' points to. */
    binTreeClass(ptrType NodePtr) ;

    /* Copies the tree rooted at TreePtr into a tree rooted at
     NewTreePtr. */
    void CopyTree(ptrType TreePtr, ptrType& NewTreePtr) const;

    /* Deallocates memory for a tree. */
    void DestroyTree(ptrType& TreePtr);

    // The next two functions retrieve and set the value
    // of the private data member Root.

    /* Return the root pointer of the tree. */
    ptrType RootPtr() const;

    /* Set the root pointer of the tree to 'NewRoot'. */
    void SetRootPtr(ptrType NewRoot);

    // The next two functions retrieve and set the values
```

```
      // of the left and right child pointers of a tree node.

      /* Follow NodePtr to the node, X, to which it points.  Set
     LChildPtr equal to the left child pointer of X.  Also
     set RChildPtr equal to the right child pointer of X. */
    void GetChildPtrs(ptrType NodePtr, ptrType& LChildPtr,
                      ptrType& RChildPtr) const;

      /* Follow NodePtr to the node, X, to which it points. Set
     the left child pointer of X to LChildPtr, and the right
     child pointer of X to RChildPtr. */
    void SetChildPtrs(ptrType NodePtr, ptrType LChildPtr,
                      ptrType RChildPtr);

      /* Traverse, in pre-order, the tree to which 'TreePtr'
     points.  While traversing, apply 'Visit' to each node.*/
    void Preorder(ptrType TreePtr, functionType Visit);

      /* Traverse, in in-order, the tree to which 'TreePtr'
     points.  While traversing, apply 'Visit' to each node.*/
    void Inorder(ptrType TreePtr, functionType Visit);

      /* Traverse, in post-order, the tree to which 'TreePtr' points.
         While traversing, apply 'Visit' to each node. */
    void Postorder(ptrType TreePtr, functionType Visit);

  private:
    ptrType Root;  // pointer to root of tree
};  // end class
// End of header file.
```