```
// ---- File 1 follows ----
// File FHtree.h
// Template definitions for FHtrees, which are general trees
#ifndef FHTREE_H
#define FHTREE_H
#include <string>

// advanced prototype for the FHtreeNode to use to declare a friend
template <class Object>
class FHtree;
```

21

```
// ---------------------- FHtreeNode Prototype --------------------------
template <class Object>
class FHtreeNode
{
   friend class FHtree<Object>;

protected:
   FHtreeNode *firstChild, *sib, *prev;
   Object data;
   FHtreeNode *myRoot;  // needed to test for certain error

public:
   FHtreeNode(const Object & d = Object(),
      FHtreeNode *sb = NULL, FHtreeNode *chld = NULL, FHtreeNode *prv = NULL)
      : firstChild(chld), sib(sb), prev(prv), data(d), myRoot(NULL)
   { }
   Object GetData() const { return data; }

protected:
   // for use only by FHtree
   FHtreeNode(const Object & d,
      FHtreeNode *sb, FHtreeNode *chld, FHtreeNode *prv,
      FHtreeNode *root)
      : firstChild(chld), sib(sb), prev(prv), data(d), myRoot(root)
   { }
};
```

good

```
// --------------------------- FHtree Prototype ------------------------------
template <class Object>
class FHtree
{
protected:
   int mSize;
   FHtreeNode<Object> *mRoot;

public:
   FHtree() { mSize = 0; mRoot = NULL; }
   FHtree(const FHtree &rhs) { mRoot = NULL; mSize = 0; *this = rhs; }
   virtual ~FHtree() { clear(); }
   bool empty() const { return (mSize == 0); }
   int size() const { return mSize; }
   void clear() { removeNode(mRoot); }
   const FHtree & operator=(const FHtree &rhs);

   FHtreeNode<Object> *addChild(FHtreeNode<Object> *treeNode, const Object &x);

   FHtreeNode<Object> *find(const Object &x) { return find(mRoot, x); }
   FHtreeNode<Object> *find(FHtreeNode<Object> *root,
      const Object &x, int level = 0);

   bool remove(const Object &x) { return remove(mRoot, x); }
   bool remove(FHtreeNode<Object> *root, const Object &x);
   void removeNode(FHtreeNode<Object> *nodeToDelete);

   // usual client interfaces (entire tree implied)
   void display() const { display(mRoot, 0); }
   template <class Processor>
   void traverse(Processor func) const { traverse(func, mRoot, 0); }

   // recursive helpers
   void display(FHtreeNode<Object> *treeNode, int level = 0) const;
   template <class Processor>
   void traverse(Processor func, FHtreeNode<Object> *treeNode, int level = 0)
      const;

protected:
   FHtreeNode<Object> *clone(FHtreeNode<Object> *root) const;
   void setMyRoots(FHtreeNode<Object> *treeNode);
};

// FHtree Method Definitions -------------------------------------------------
template <class Object>
FHtreeNode<Object>* FHtree<Object>::find(FHtreeNode<Object> *root,
   const Object &x, int level)
{
   FHtreeNode<Object> *retval;

   if (mSize == 0 || root == NULL)
      return NULL;

   if (root->data == x)
      return root;

   // otherwise, recurse.  don't process sibs if this was the original call
   if (level > 0 && (retval = find(root->sib, x, level)))
      return retval;
   return find(root->firstChild, x, level + 1);
}

template <class Object>
bool FHtree<Object>::remove(FHtreeNode<Object> *root, const Object &x)
{
   FHtreeNode<Object> *tn = NULL;

   if (mSize == 0 || root == NULL)
      return false;

   if ((tn = find(root, x)) != NULL)
   {
      removeNode(tn);
      return true;
   }
   return false;
}

template <class Object>
const FHtree<Object> &FHtree<Object>::operator=
(const FHtree &rhs)
{
   if (&rhs != this)
   {
      clear();
      mRoot = clone(rhs.mRoot);
      mSize = rhs.mSize;
      setMyRoots(mRoot);
   }
   return *this;
}

template <class Object>
void FHtree<Object>::removeNode(FHtreeNode<Object> *nodeToDelete)
{
   if (nodeToDelete == NULL || mRoot == NULL)
      return;
   if (nodeToDelete->myRoot != mRoot)
      return;  // silent error, node does not belong to this tree

               // remove all the children of this node
   while (nodeToDelete->firstChild)
      removeNode(nodeToDelete->firstChild);

   if (nodeToDelete->prev == NULL)
      mRoot = NULL;  // last node in tree
   else if (nodeToDelete->prev->sib == nodeToDelete)
      nodeToDelete->prev->sib = nodeToDelete->sib; // adjust left sibling
   else
      nodeToDelete->prev->firstChild = nodeToDelete->sib;  // adjust parent

                                                           // adjust the success
or sib's prev pointer
   if (nodeToDelete->sib != NULL)
      nodeToDelete->sib->prev = nodeToDelete->prev;

   delete nodeToDelete;
   --mSize;
}

template <class Object>
FHtreeNode<Object> *FHtree<Object>::addChild(
   FHtreeNode<Object> *treeNode, const Object &x)
{
   // empty tree? - create a root node if user passes in NULL
   if (mSize == 0)
   {
      if (treeNode != NULL)
         return NULL; // silent error something's fishy.  treeNode can't right
      mRoot = new FHtreeNode<Object>(x, NULL, NULL, NULL);
      mRoot->myRoot = mRoot;
      mSize = 1;
      return mRoot;
   }
   if (treeNode == NULL)
      return NULL; // silent error inserting into a non_null tree with a null pa
rent
   if (treeNode->myRoot != mRoot)
      return NULL;  // silent error, node does not belong to this tree

                    // push this node into the head of the sibling list; adjust
prev pointers
   FHtreeNode<Object> *newNode = new FHtreeNode<Object>(x,
      treeNode->firstChild, NULL, treeNode, mRoot);  // sib, child, prev, root
   treeNode->firstChild = newNode;
   if (newNode->sib != NULL)
      newNode->sib->prev = newNode;
   ++mSize;
   return newNode;
}

template <class Object>
void FHtree<Object>::display(FHtreeNode<Object> *treeNode, int level) const
{
   // this will be static and so will be shared by all calls - a special techniq
ue to
   // be avoided in recursion, usually
   static string blankString = "                                    ";
   string indent;

   // stop runaway indentation/recursion
   if (level > (int)blankString.length() - 1)
   {
      cout << blankString << " ... " << endl;
      return;
   }

   if (treeNode == NULL)
      return;

   indent = blankString.substr(0, level);

   cout << indent << treeNode->data << endl;
   display(treeNode->firstChild, level + 1);
   if (level > 0)
      display(treeNode->sib, level);
}

template <class Object>
template <class Processor>
void FHtree<Object>::traverse(Processor func, FHtreeNode<Object> *treeNode, int
level)
const
{
   if (treeNode == NULL)
      return;

   func(treeNode->data);

   traverse(func, treeNode->firstChild, level + 1);
   if (level > 0)
      traverse(func, treeNode->sib, level);
}

template <class Object>
FHtreeNode<Object> *FHtree<Object>::clone(
   FHtreeNode<Object> *root) const
{
   FHtreeNode<Object> *newNode;
   if (root == NULL)
      return NULL;

   // does not set myRoot which must be done by caller
   newNode = new FHtreeNode<Object>(
      root->data,
      clone(root->sib), clone(root->firstChild));

   // entire subtree is cloned, but wire this node into its sib and first chld
   if (newNode->sib)
      newNode->sib->prev = newNode;
   if (newNode->firstChild)
      newNode->firstChild->prev = newNode;
   return newNode;
}

template <class Object>
void FHtree<Object>::setMyRoots(FHtreeNode<Object> *treeNode)
{
   if (treeNode == NULL)
      return;

   treeNode->myRoot = mRoot;
   setMyRoots(treeNode->sib);
   setMyRoots(treeNode->firstChild);
}

#endif
```

even if you use provided code, make sure it's reformatted to our spec

```
// ---- File 2 follows ----
#include "FHtree.h"
template <class Object>
class FHsdTreeNode : public FHtreeNode<Object>
{
private:
   bool deleted;
public:
   FHsdTreeNode(const Object & d = Object(),
                FHsdTreeNode *sb = NULL, FHsdTreeNode *chld = NULL,
                FHsdTreeNode *prv = NULL, FHsdTreeNode* root = NULL, bool delete
d = false);
   FHsdTreeNode* getFirstChild() const { return (FHsdTreeNode*)FHtreeNode<Object
>::firstChild; }
   void setFirstChild(const FHsdTreeNode* firstChild) { this->firstChild = (FHsd
TreeNode*)firstChild; }
   FHsdTreeNode* getPrev() const { return (FHsdTreeNode<Object>*)FHtreeNode<Obje
ct>::prev; }
   void setPrev(const FHsdTreeNode* prev) { this->prev = (FHsdTreeNode*)prev; }
   FHsdTreeNode* getMyRoot() const { return (FHsdTreeNode*)FHtreeNode<Object>::m
yRoot; }
   void setMyRoot(FHsdTreeNode* myRoot) { this->myRoot = myRoot; }
   FHsdTreeNode* getSib() const { return (FHsdTreeNode*)FHtreeNode<Object>::sib;
 }
   void setSib(const FHsdTreeNode* sib) { this->sib = (FHsdTreeNode<Object>*)sib
; }


   Object getData() const { return FHtreeNode<Object>::data; }
   void setData(const Object& object) { this->data = object; }

   bool getDeleted() const { return deleted; }
   void setDeleted(bool deleted) { this->deleted = deleted; }
};

template <class Object>
FHsdTreeNode<Object>::FHsdTreeNode(const Object & d,
                                   FHsdTreeNode *sb, FHsdTreeNode *chld, FHsdTre
eNode *prv, FHsdTreeNode* root,
                                   bool deleted) : FHtreeNode<Object>(d, sb, chl
d, prv, root)
{
   this->deleted = deleted;
}
```

cool

```
template <class Object>
class FHsdTree : public FHtree<Object>
{
public:
   FHsdTree() : FHtree<Object>() {}
   FHsdTree(const FHsdTree<Object> &rhs) : FHtree<Object>(rhs) {}
   ~FHsdTree() { clear(); }
   int size() const { return size((FHsdTreeNode<Object>*)FHtree<Object>::mRoot,
0); }
   bool empty() const { return (size() == 0); }
   int sizePhysical() const { return FHtree<Object>::mSize; }
   void displayPhysical() const { displayPhysical((FHsdTreeNode<Object>*)FHtree<
Object>::mRoot); }
   void displayPhysical(FHsdTreeNode<Object> *treeNode, int level = 0) const;
   void clear() { removeNode((FHsdTreeNode<Object>*)FHtree<Object>::mRoot); }
   bool collectGarbage() { return collectGarbage((FHsdTreeNode<Object>*)FHtree<O
bject>::mRoot, 0); };
   bool collectGarbage(FHsdTreeNode<Object> *treeNode, int level = 0);

   // TODO
   const FHsdTree & operator=(const FHsdTree &rhs);

   FHsdTreeNode<Object> *addChild(FHsdTreeNode<Object> *treeNode, const Object &
x);

   FHsdTreeNode<Object> *find(const Object &x) { return find((FHsdTreeNode<Objec
t>*)FHtree<Object>::mRoot, x); }
   FHsdTreeNode<Object> *find(FHsdTreeNode<Object> *root,
                             const Object &x, int level = 0);

   bool remove(const Object &x) { return remove((FHsdTreeNode<Object>*)FHtree<Ob
ject>::mRoot, x); }
   bool remove(FHsdTreeNode<Object> *root, const Object &x);
   void removeNode(FHsdTreeNode<Object> *nodeToDelete);

   // usual client interfaces (entire tree implied)
   void display() const { display((FHsdTreeNode<Object>*)FHtree<Object>::mRoot);
 }
   template <class Processor>
   void traverse(Processor func) const { traverse(func, (FHsdTreeNode<Object>*)F
Htree<Object>::mRoot, 0); }

   // recursive helpers
   void display(FHsdTreeNode<Object> *treeNode, int level = 0) const;
   int size(FHsdTreeNode<Object> *treeNode, int level = 0) const;
   template <class Processor>
   void traverse(Processor func, FHsdTreeNode<Object> *treeNode, int level = 0)
   const;

protected:
   FHsdTreeNode<Object> *clone(FHsdTreeNode<Object> *root) const;
   void setMyRoots(FHsdTreeNode<Object> *treeNode);
};

// FHsdTree Method Definitions -------------------------------------------------
template <class Object>
FHsdTreeNode<Object>* FHsdTree<Object>::find(FHsdTreeNode<Object> *root,
                                             const Object &x, int level)
{
   FHsdTreeNode<Object> *retval;
   if (FHtree<Object>::mSize == 0 || root == NULL)
      return NULL;
   if (root->getDeleted())
      return NULL;
   if (root->getData() == x)
      return root;

   // otherwise, recurse.  don't process sibs if this was the original call
   if (level > 0 && (retval = find(root->getSib(), x, level)))
      return retval;
   return find(root->getFirstChild(), x, level + 1);
}

template <class Object>
bool FHsdTree<Object>::remove(FHsdTreeNode<Object> *root, const Object &x)
{
   FHsdTreeNode<Object> *tn = NULL;

   if (FHtree<Object>::mSize == 0 || root == NULL)
      return false;

   if ((tn = find(root, x)) != NULL)
   {
      tn->setDeleted(true);
      return true;
   }
   return false;
}

template <class Object>
const FHsdTree<Object> &FHsdTree<Object>::operator=
(const FHsdTree &rhs) {
   *this = (FHtree<Object>*)rhs;
}

template <class Object>
void FHsdTree<Object>::removeNode(FHsdTreeNode<Object> *nodeToDelete)
{
   if (nodeToDelete == NULL || FHtree<Object>::mRoot == NULL)
      return;
   if (nodeToDelete->getMyRoot() != FHtree<Object>::mRoot)
      return;  // silent error, node does not belong to this tree

   // remove all the children of this node
   while (nodeToDelete->getFirstChild())
      removeNode(nodeToDelete->getFirstChild());

   if (nodeToDelete->getPrev() == NULL)
      FHtree<Object>::mRoot = NULL;  // last node in tree
   else if (nodeToDelete->getPrev()->getSib() == nodeToDelete)
      nodeToDelete->getPrev()->setSib(nodeToDelete->getSib()); // adjust left si
bling
   else
      nodeToDelete->getPrev()->setFirstChild(nodeToDelete->getSib());  // adjust
 parent

   // adjust the successor sib's prev pointer
   if (nodeToDelete->getSib() != NULL)
      nodeToDelete->getSib()->setPrev(nodeToDelete->getPrev());

   delete nodeToDelete;
   --FHtree<Object>::mSize;
```

```
}


template <class Object>
FHsdTreeNode<Object> *FHsdTree<Object>::addChild(
                                                 FHsdTreeNode<Object> *treeNode,
 const Object &x)
{
   // empty tree? - create a root node if user passes in NULL
   if (FHtree<Object>::mSize == 0)
   {
      if (treeNode != NULL)
         return NULL; // silent error something's fishy.  treeNode can't right
      FHtree<Object>::mRoot = new FHsdTreeNode<Object>(x, NULL, NULL, NULL);

      ((FHsdTreeNode<Object>*)FHtree<Object>::mRoot)->setMyRoot((FHsdTreeNode<Ob
ject>*)FHtree<Object>::mRoot);
      FHtree<Object>::mSize = 1;
      return (FHsdTreeNode<Object>*)FHtree<Object>::mRoot;
   }
   if (treeNode == NULL)
      return NULL; // silent error inserting into a non_null tree with a null pa
rent
   if (treeNode->getMyRoot() != FHtree<Object>::mRoot)
      return NULL;  // silent error, node does not belong to this tree
   if (treeNode->getDeleted())
      return NULL;
   // push this node into the head of the sibling list; adjust prev pointers
   FHsdTreeNode<Object> *newNode = new FHsdTreeNode<Object>(x,
                                                            treeNode->getFirstCh
ild(), NULL, treeNode, (FHsdTreeNode<Object>*)FHtree<Object>::mRoot);  // sib, c
hild, prev, root
   treeNode->setFirstChild(newNode);
   if (newNode->getSib() != NULL)
      newNode->getSib()->setPrev(newNode);
   ++FHtree<Object>::mSize;
   return newNode;
}

template <class Object>
void FHsdTree<Object>::display(FHsdTreeNode<Object> *treeNode, int level) const
{
   // this will be static and so will be shared by all calls - a special techniq
ue to
   // be avoided in recursion, usually
   static string blankString = "                                     ";
   string indent;

   // stop runaway indentation/recursion
   if (level > (int)blankString.length() - 1)
   {
      cout << blankString << " ... " << endl;
      return;
   }

   if (treeNode == NULL)
      return;

   if (!treeNode->getDeleted())
   {
      indent = blankString.substr(0, level);
      cout << indent << treeNode->getData() << endl;
      display(treeNode->getFirstChild(), level + 1);
   }

   if (level > 0)
      display(treeNode->getSib(), level);
}

template <class Object>
template <class Processor>
void FHsdTree<Object>::traverse(Processor func, FHsdTreeNode<Object> *treeNode,
int level)
const
{
   if (treeNode == NULL)
      return;

   if (!treeNode->getDeleted())
   {
      func(treeNode->getData());
      traverse(func, treeNode->getFirstChild(), level + 1);
   }

   if (level > 0)
      traverse(func, treeNode->getSib(), level);
}

template <class Object>
FHsdTreeNode<Object> *FHsdTree<Object>::clone(
                                              FHsdTreeNode<Object> *root) const
{
   FHsdTreeNode<Object> *newNode;
   if (root == NULL)
      return NULL;

   // does not set myRoot which must be done by caller
   newNode = new FHsdTreeNode<Object>(
                                      root->getData(),
                                      clone(root->getSib()), clone(root->getFirs
tChild()));

   // entire subtree is cloned, but wire this node into its sib and first chld
   if (newNode->getSib())
      newNode->getSib()->setPrev(newNode);
   if (newNode->getFirstChild())
      newNode->getFirstChild.setPrev(newNode);
   return newNode;
}

template <class Object>
void FHsdTree<Object>::setMyRoots(FHsdTreeNode<Object> *treeNode)
{
   if (treeNode == NULL)
      return;

   treeNode->setMyRoot(FHtree<Object>::mRoot);
   setMyRoots(treeNode->getSib());
   setMyRoots(treeNode->getFirstChild());
}

template <class Object>
int FHsdTree<Object>::size(FHsdTreeNode<Object> *treeNode, int level) const
{
   if (treeNode == NULL)
      return 0;

   int numNode = 0;

   if (level > 0)
      numNode += size(treeNode->getSib(), level);

   if (!treeNode->getDeleted())
   {
      numNode += size(treeNode->getFirstChild(), level + 1);
      numNode++;
   }

   return numNode;
}

template <class Object>
void FHsdTree<Object>::displayPhysical(FHsdTreeNode<Object> *treeNode, int level
) const
{
   // this will be static and so will be shared by all calls - a special techniq
ue to
   // be avoided in recursion, usually
   static string blankString = "                                        ";
   string indent;

   // stop runaway indentation/recursion
   if (level > (int)blankString.length() - 1)
   {
      cout << blankString << " ... " << endl;
      return;
   }

   if (treeNode == NULL)
      return;

   indent = blankString.substr(0, level);
   cout << indent << treeNode->getData();
   if (treeNode->getDeleted())
      cout << " (D)";
   cout << endl;
   displayPhysical(treeNode->getFirstChild(), level + 1);


   if (level > 0)
      displayPhysical(treeNode->getSib(), level);
}

template <class Object>
bool FHsdTree<Object>::collectGarbage(FHsdTreeNode<Object> *treeNode, int level)
{
   if (treeNode == NULL)
      return false;

   bool isChildDeleted = false, isSibDeleted = false;
   bool isNodeDeleted = treeNode->getDeleted();
   if (isNodeDeleted)
   {
      removeNode(treeNode);
   }
   else
```

-2

remove sibs first if necessary

```
   {
      isChildDeleted = collectGarbage(treeNode->getFirstChild(), level + 1);
   }

   if (level > 0) {
      isSibDeleted = collectGarbage(treeNode->getSib(), level);
   }

   return (isSibDeleted || isChildDeleted || isNodeDeleted);
}

// ---- File 3 follows ----
#include <iostream>
#include <string>
using namespace std;
#include "SoftDelTree.h"
int main()
{
   FHsdTree<string> sceneTree;
   FHsdTreeNode<string> *tn;
   cout << "Starting tree empty? " << sceneTree.empty() << endl << endl;
   // create a scene in a room
   tn = sceneTree.addChild(NULL, "room");
   // add three objects to the scene tree
   sceneTree.addChild(tn, "Lily the canine");
   sceneTree.addChild(tn, "Miguel the human");
   sceneTree.addChild(tn, "table");
   // add some parts to Miguel
   tn = sceneTree.find("Miguel the human");
   // Miguel's left arm
   tn = sceneTree.addChild(tn, "torso");
   tn = sceneTree.addChild(tn, "left arm");
   tn = sceneTree.addChild(tn, "left hand");
   sceneTree.addChild(tn, "thumb");
   sceneTree.addChild(tn, "index finger");
   sceneTree.addChild(tn, "middle finger");
   sceneTree.addChild(tn, "ring finger");
   sceneTree.addChild(tn, "pinky");
   // Miguel's right arm
   tn = sceneTree.find("Miguel the human");
   tn = sceneTree.find(tn, "torso");
   tn = sceneTree.addChild(tn, "right arm");
   tn = sceneTree.addChild(tn, "right hand");
   sceneTree.addChild(tn, "thumb");
   sceneTree.addChild(tn, "index finger");
   sceneTree.addChild(tn, "middle finger");
   sceneTree.addChild(tn, "ring finger");
   sceneTree.addChild(tn, "pinky");
   // add some parts to Lily
   tn = sceneTree.find("Lily the canine");
   tn = sceneTree.addChild(tn, "torso");
   sceneTree.addChild(tn, "right front paw");
   sceneTree.addChild(tn, "left front paw");
   sceneTree.addChild(tn, "right rear paw");
   sceneTree.addChild(tn, "left rear paw");
   sceneTree.addChild(tn, "spare mutant paw");
   sceneTree.addChild(tn, "wagging tail");
   // add some parts to table
   tn = sceneTree.find("table");
   sceneTree.addChild(tn, "north east leg");
   sceneTree.addChild(tn, "north west leg");
   sceneTree.addChild(tn, "south east leg");
   sceneTree.addChild(tn, "south west leg");
   cout << "\n------------ Loaded Tree ------------------ \n";
   sceneTree.display();
   sceneTree.remove("spare mutant paw");
   sceneTree.remove("Miguel the human");
   sceneTree.remove("an imagined higgs boson");
   cout << "\n------------ Virtual (soft) Tree ----------------- \n";
   sceneTree.display();
   cout << "\n------------ Physical (hard) Display ------------------ \n";
   sceneTree.displayPhysical();
   cout << "------- Testing Sizes (compare with above)----------- \n";
   cout << "virtual (soft) size: " << sceneTree.size() << endl;
   cout << "physical (hard) size: " << sceneTree.sizePhysical() << endl;

   cout << "------------ Collecting Garbage -------------------- \n";
   cout << "found soft-deleted nodes? " << sceneTree.collectGarbage() << endl;
   cout << "immediate collect again? " << sceneTree.collectGarbage() << endl;
   cout << "--------- Hard Display after garb col ------------ \n";
   sceneTree.displayPhysical();
   cout << "Semi-deleted tree empty? " << sceneTree.empty() << endl << endl;
   sceneTree.remove("room");
   cout << "Completely-deleted tree empty? " << sceneTree.empty() << endl << end
l;
   return 0;
}


/*---------------------------- Posted Run -------------------------------------
-
Starting tree empty? 1


------------ Loaded Tree -----------------
room
 table
  south west leg
  south east leg
  north west leg
  north east leg
 Miguel the human
  torso
   right arm
    right hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
   left arm
    left hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
 Lily the canine
  torso
   wagging tail
   spare mutant paw
   left rear paw
   right rear paw
   left front paw
   right front paw

------------ Virtual (soft) Tree -----------------
room
 table
  south west leg
  south east leg
  north west leg
  north east leg
 Lily the canine
  torso
   wagging tail
   left rear paw
   right rear paw
   left front paw
   right front paw

------------ Physical (hard) Display -----------------
room
 table
  south west leg
  south east leg
  north west leg
  north east leg
 Miguel the human (D)
  torso
   right arm
    right hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
   left arm
    left hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
 Lily the canine
  torso
   wagging tail
   spare mutant paw (D)
   left rear paw
   right rear paw
   left front paw
   right front paw
------- Testing Sizes (compare with above)-----------
virtual (soft) size: 13
physical (hard) size: 30
------------ Collecting Garbage --------------------
found soft-deleted nodes? 1
immediate collect again? 0
--------- Hard Display after garb col ------------
room
 table
  south west leg
  south east leg
  north west leg
  north east leg
 Lily the canine
```

```
  torso
   wagging tail
   left rear paw
   right rear paw
   left front paw
   right front paw
Semi-deleted tree empty? 0

Completely-deleted tree empty? 1
Press any key to continue . . .
--------------------------------------------------------------------------------
-*/
```

```
// CS 2B Lab 9 - Lazy Deletion in Trees
// Instructor Solution:
//  Original - Prof. Loceff, Updates, Edits, Annotations:&
//
// Notes:
//  - Multiple files (submitted as one file): SoftDelTree.h and main.cpp
//  - Faithfulness to spec
//  - Correct method qualifications (including virtuals)
//  - Correct handling of lazy deletion

// -----------------------------------------------------------------------
//  File SoftDelTree.h
//  Template definitions for FHsdTrees, which are general trees
//
#ifndef SoftDelTree_h
#define SoftDelTree_h

#include <string>

// advanced prototype for the FHsdTreeNode to use to declare a friend
template <class T>
class FHsdTree;

// ----- FHsdTreeNode Prototype -----
template <class T>
class FHsdTreeNode {
    friend class FHsdTree<T>;

protected:
    FHsdTreeNode *firstChild, *sib, *prev;
    T data;
    FHsdTreeNode *myRoot;  // needed to test for certain error
    bool deleted;

public:
    FHsdTreeNode(const T& d = T(),
                 FHsdTreeNode *sb = NULL,
                 FHsdTreeNode *child = NULL,
                 FHsdTreeNode *prv = NULL,
                 bool isDeleted = false)
    : firstChild(child), sib(sb), prev(prv), data(d), myRoot(NULL), deleted(isDeleted)
    { }
    T getData() const { return data; }

protected:
    // for use only by FHsdTree
    FHsdTreeNode(const T& d,
                 FHsdTreeNode *sb, FHsdTreeNode *child, FHsdTreeNode *prv,
                 FHsdTreeNode *root, bool isDeleted)
        : firstChild(child), sib(sb), prev(prv), data(d), myRoot(root),
          deleted(isDeleted)
    { }
};
```

```
// ----- FHsdTree Prototype -----
template <class T>
class FHsdTree {
protected:
    int mSize;
    FHsdTreeNode<T> *mRoot;

public:
    FHsdTree() { mSize = 0; mRoot = NULL; }
    FHsdTree(const FHsdTree &rhs) { mRoot = NULL; mSize = 0; *this = rhs; }
    virtual ~FHsdTree() { clear(); }

    bool empty() const;
    int sizePhysical() const { return mSize; }
    int size() const { return size(mRoot); }
    int size(FHsdTreeNode<T> *root) const;
    void clear() { removeNode(mRoot); }
    const FHsdTree& operator=(const FHsdTree& rhs);

    FHsdTreeNode<T> *addChild(FHsdTreeNode<T> *treeNode, const T &x);

    FHsdTreeNode<T> *find(const T &x) { return find(mRoot, x); }
    FHsdTreeNode<T> *find(FHsdTreeNode<T> *root, const T &x, int level = 0);

    bool remove(const T &x) { return remove(mRoot, x); }
    bool remove(FHsdTreeNode<T> *root, const T& x);
    void removeNode(FHsdTreeNode<T> *nodeToDelete);

    // usual client interfaces (entire tree implied)
    void display() const { display(mRoot, 0); }
    void displayPhysical() const { displayPhysical(mRoot, 0); }
    template <class F> void traverse(F func) const {
        traverse(func, mRoot, 0);
    }

    // recursive helpers
    void display(FHsdTreeNode<T> *treeNode, int level = 0) const;
    void displayPhysical(FHsdTreeNode<T> *treeNode, int level = 0) const;
    template <class F> void traverse(F func, FHsdTreeNode<T> *treeNode,
                                     int level = 0) const;

    bool collectGarbage() { return collectGarbage(mRoot); }
    bool collectGarbage(FHsdTreeNode<T> *treeNode);

protected:
    FHsdTreeNode<T> *clone(FHsdTreeNode<T> *root) const;
    void setMyRoots(FHsdTreeNode<T> *treeNode);
};
```

```
// ----- FHsdTree Method Definitions -----
template <class T>
FHsdTreeNode<T>* FHsdTree<T>::find(FHsdTreeNode<T> *root, const T& x, int level) {
    FHsdTreeNode<T> *retval;

    if (mSize == 0 || root == NULL)
        return NULL;

    if (root->data == x && !root->deleted)
        return root;

    // otherwise, recurse. Don't process sibs if this was the original call
    if (level > 0 && (retval = find(root->sib, x, level)))
        return retval;

    // don't process children if this root is deleted
    if (root->deleted)
        return NULL;

    return find(root->firstChild, x, ++level);
}

template <class T>
bool FHsdTree<T>::remove(FHsdTreeNode<T> *root, const T& x) {
    FHsdTreeNode<T> *tn = NULL;

    if (mSize == 0 || root == NULL)
        return false;

    if ( (tn = find(root, x)) != NULL ) {
        tn->deleted = true;
        return true;
    }
    return false;
}

template <class T> const FHsdTree<T>&
FHsdTree<T>::operator=(const FHsdTree &rhs) {
    if (&rhs != this) {
        clear();
        mRoot = clone(rhs.mRoot);
        mSize = rhs.mSize;
        setMyRoots(mRoot);
    }
    return *this;
}
```

```
template <class T>
void FHsdTree<T>::removeNode(FHsdTreeNode<T> *nodeToDelete) {
    if (nodeToDelete == NULL || mRoot == NULL)
        return;
    if (nodeToDelete->myRoot != mRoot)
        return;  // silent error, node does not belong to this tree

    // remove all the children of this node
    while (nodeToDelete->firstChild)
        removeNode(nodeToDelete->firstChild);

    if (nodeToDelete->prev == NULL)
        mRoot = NULL;  // last node in tree
    else if (nodeToDelete->prev->sib == nodeToDelete)
        nodeToDelete->prev->sib = nodeToDelete->sib; // adjust left sibling
    else
        nodeToDelete->prev->firstChild = nodeToDelete->sib;  // adjust parent

    // adjust the successor sib's prev pointer
    if (nodeToDelete->sib != NULL)
        nodeToDelete->sib->prev = nodeToDelete->prev;

    delete nodeToDelete;
    --mSize;
}

template <class T>
FHsdTreeNode<T> *FHsdTree<T>::addChild(FHsdTreeNode<T> *treeNode, const T& x) {
    // empty tree? - create a root node if user passes in NULL
    if (mSize == 0) {
        if (treeNode != NULL)
            return NULL; // Silent error. something's fishy. treeNode can't right
        mRoot = new FHsdTreeNode<T>(x, NULL, NULL, NULL);
        mRoot->myRoot = mRoot;
        mSize = 1;
        return mRoot;
    }
    if (treeNode == NULL)
        return NULL; // silent error inserting into non_null tree w/null parent
    if (treeNode->myRoot != mRoot)
        return NULL;  // silent error, node does not belong to this tree

    // push this node into the head of the sibling list; adjust prev pointers
    // parms: sb, child, prv, rt
    FHsdTreeNode<T> *newNode = new FHsdTreeNode<T>(x,
                                                   treeNode->firstChild, NULL,
                                                   treeNode, mRoot, false);
    treeNode->firstChild = newNode;
    if (newNode->sib != NULL)
        newNode->sib->prev = newNode;
    ++mSize;
    return newNode;
}
```

```
template <class T>
void FHsdTree<T>::displayPhysical(FHsdTreeNode<T> *treeNode, int level) const {
    // this is static and so will be shared by all calls -special technique to
    // be avoided in recursion, usually
    static string blankString = "                                  ";
    string indent;

    // stop runaway indentation/recursion
    if (level > (int) blankString.length() - 1) {
        cout << blankString << " ... " << endl;
        return;
    }

    if (treeNode == NULL)
        return;

    indent = blankString.substr(0, level);

    cout << indent
         << treeNode->data
         << (treeNode->deleted? " (D)" : "")
         << endl;
    displayPhysical(treeNode->firstChild, level + 1);

    if (level > 0)
        displayPhysical( treeNode->sib, level );
}

template <class T>
void FHsdTree<T>::display(FHsdTreeNode<T> *treeNode, int level) const {
    // this will be static and so will be shared by all calls
    static string blankString = "                                  ";
    string indent;

    // stop runaway indentation/recursion
    if (level > (int) blankString.length() - 1) {
        cout << blankString << " ... " << endl;
        return;
    }

    indent = blankString.substr(0, level);

    if (treeNode == NULL)
        return;

    // direct case: detect a soft-deleted node
    if (!treeNode->deleted) {
        cout << indent << treeNode->data  << endl;
        // recurse down
        display( treeNode->firstChild, level + 1 );
    }
    // recurse right
    if (level > 0)
        display( treeNode->sib, level );
}

template <class T>
template <class F>
void FHsdTree<T>::traverse(F func, FHsdTreeNode<T> *treeNode, int level) const {
    if (treeNode == NULL)
        return;

    // direct case: detect a soft-deleted node
    if (!treeNode->deleted) {
        func(treeNode->data);
        // recurse down
        traverse(func, treeNode->firstChild, level+1);
    }

    // recurse right
    if (level > 0)
        traverse(func, treeNode->sib, level);
}

template <class T>
FHsdTreeNode<T> *FHsdTree<T>::clone(FHsdTreeNode<T> *root) const {
    FHsdTreeNode<T> *newNode;
    if (root == NULL)
        return NULL;

    // does not set myRoot which must be done by caller
    newNode = new FHsdTreeNode<T>(root->data,
                                  clone(root->sib),
                                  clone(root->firstChild));
    if (newNode->sib)
        newNode->sib->prev = newNode;
    if (newNode->firstChild)
        newNode->firstChild->prev = newNode;
    newNode->deleted = root->deleted;

    return newNode;
}

template <class T>
void FHsdTree<T>::setMyRoots(FHsdTreeNode<T> *treeNode) {
    FHsdTreeNode<T> *child;

    if (mRoot == NULL)
        return;

    treeNode->myRoot = mRoot;
    for (child = treeNode->firstChild; child != NULL; child = child->sib)
        setMyRoots(child);
}
```

```
template <class T>
int FHsdTree<T>::size(FHsdTreeNode<T> *root) const {
    int sibSize, countThis, childrenSize;

    if (root == NULL)
        return 0;

    // count siblings
    sibSize = size(root->sib);

    if(root->deleted) {
        childrenSize = 0;
        countThis = 0;
    } else {
        childrenSize = size(root->firstChild);
        countThis = 1;
    }

    return childrenSize + sibSize + countThis;
}

template <class T>
bool FHsdTree<T>::collectGarbage(FHsdTreeNode<T> *root) {
    bool sibResult = false, childrenResult = false, thisResult = false;

    if (root == NULL)
        return false;

    // collect sib garbage (must do before root removed)
    sibResult = collectGarbage( root->sib );

    if(root->deleted) {
        removeNode(root);   // will remove all children
        thisResult = true;
    } else {
        // since root not deleted, must remove children manually
        childrenResult = collectGarbage( root->firstChild);
    }

    // if anything was deleted, return true
    return sibResult ||  childrenResult || thisResult ;
}

template <class T>
bool FHsdTree<T>::empty() const {
    // hard empty?
    if (mRoot == NULL)
        return true;

    // soft empty?
    if ( mRoot->deleted )
        return true;
    return false;
}

#endif /* SoftDelTree_h */

// -----------------------------------------------------------------------------
// Test Driver
// File main.cpp

#include <iostream>
#include <string>
using namespace std;

#include "SoftDelTree.h"

int main()
{
    FHsdTree<string> sceneTree, clonedTree;
    FHsdTreeNode<string> *tn;

    cout << "Starting tree empty? " << sceneTree.empty() << endl << endl;

    // create a scene in a room
    tn = sceneTree.addChild(NULL, "room");

    // add three objects to the scene tree
    sceneTree.addChild(tn, "Lily the canine");

    sceneTree.addChild(tn, "Miguel the human");
    sceneTree.addChild(tn, "table");

    // add some parts to Miguel
    tn = sceneTree.find("Miguel the human");

    // Miguel's left arm
    tn = sceneTree.addChild(tn, "torso");
    tn = sceneTree.addChild(tn, "left arm");
    tn =  sceneTree.addChild(tn, "left hand");
    sceneTree.addChild(tn, "thumb");
    sceneTree.addChild(tn, "index finger");
    sceneTree.addChild(tn, "middle finger");
    sceneTree.addChild(tn, "ring finger");
    sceneTree.addChild(tn, "pinky");

    // Miguel's right arm
    tn = sceneTree.find("Miguel the human");
    tn = sceneTree.find(tn, "torso");
    tn = sceneTree.addChild(tn, "right arm");
    tn =  sceneTree.addChild(tn, "right hand");
    sceneTree.addChild(tn, "thumb");
    sceneTree.addChild(tn, "index finger");
    sceneTree.addChild(tn, "middle finger");
    sceneTree.addChild(tn, "ring finger");
    sceneTree.addChild(tn, "pinky");

    // add some parts to Lily
    tn = sceneTree.find("Lily the canine");
    tn = sceneTree.addChild(tn, "torso");
    sceneTree.addChild(tn, "right front paw");
    sceneTree.addChild(tn, "left front paw");
    sceneTree.addChild(tn, "right rear paw");
    sceneTree.addChild(tn, "left rear paw");
    sceneTree.addChild(tn, "spare mutant paw");
    sceneTree.addChild(tn, "wagging tail");

    // add some parts to table
    tn = sceneTree.find("table");
    sceneTree.addChild(tn, "north east leg");
    sceneTree.addChild(tn, "north west leg");
    sceneTree.addChild(tn, "south east leg");
    sceneTree.addChild(tn, "south west leg");

    cout << "\n----- Loaded Tree -----\n";
    sceneTree.display();

    sceneTree.remove("spare mutant paw");
    sceneTree.remove("Miguel the human");
    sceneTree.remove("an imagined higgs boson");

    cout << "\n----- Virtual (soft) Tree -----\n";
    sceneTree.display();

    cout << "\n----- Physical (hard) Display -----\n";
    sceneTree.displayPhysical();

    cout << "\n----- Testing Sizes (compare with above) -----\n"
         << "virtual (soft) size: " << sceneTree.size() << endl
         << "physical (hard) size: " << sceneTree.sizePhysical() << endl;

    clonedTree = sceneTree;
    cout << "\n----- Cloned Virtual (soft) Tree -----\n";
    clonedTree.display();

    cout << "\n----- Cloned Physical (hard) Display -----\n";
    clonedTree.displayPhysical();

    cout << "\n----- Cloned Sizes (compare with above) -----\n"
         << "virtual (soft) size: " << clonedTree.size() << endl
         << "physical (hard) size: " << clonedTree.sizePhysical() << endl;

    cout << "\n----- Collecting Garbage -----\n"
         << "found soft-deleted nodes? " << sceneTree.collectGarbage() << endl
         << "immediate collect again? " << sceneTree.collectGarbage() << endl;

    cout << "\n----- Hard Display after garb col -----\n";
```

```
    sceneTree.displayPhysical();

    cout << "Semi-deleted tree empty? " << sceneTree.empty() << endl << endl;
    sceneTree.remove("room");
    cout << "Completely-deleted tree empty? " << sceneTree.empty() << endl << endl;

    cout << "\n----- Cloned Physical Display Not Affected? -----\n";
    clonedTree.displayPhysical();

    return 0;
}

/* --------------------------- run ----------------------------------
Starting tree empty? 1


----- Loaded Tree -----
room
 table
  south west leg
  south east leg
  north west leg
  north east leg
 Miguel the human
  torso
   right arm
    right hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
   left arm
    left hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
 Lily the canine
  torso
   wagging tail
   spare mutant paw
   left rear paw
   right rear paw
   left front paw
   right front paw

----- Virtual (soft) Tree -----
room
 table
  south west leg
  south east leg
  north west leg
  north east leg
 Lily the canine
  torso
   wagging tail
   left rear paw
   right rear paw
   left front paw
   right front paw

----- Physical (hard) Display -----
room
 table
  south west leg
  south east leg
  north west leg
  north east leg
 Miguel the human (D)
  torso
   right arm
    right hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
   left arm
    left hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
 Lily the canine
  torso
   wagging tail
   spare mutant paw (D)
   left rear paw
   right rear paw
   left front paw
   right front paw

----- Testing Sizes (compare with above) -----
virtual (soft) size: 13
physical (hard) size: 30

----- Cloned Virtual (soft) Tree -----
room
 table
  south west leg
  south east leg
  north west leg
  north east leg
 Lily the canine
  torso
   wagging tail
   left rear paw
   right rear paw
   left front paw
   right front paw

----- Cloned Physical (hard) Display -----
room
 table
  south west leg
  south east leg
  north west leg
  north east leg
 Miguel the human (D)
  torso
   right arm
    right hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
   left arm
    left hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
 Lily the canine
  torso
   wagging tail
   spare mutant paw (D)
   left rear paw
   right rear paw
   left front paw
   right front paw

----- Cloned Sizes (compare with above) -----
virtual (soft) size: 13
physical (hard) size: 30

----- Collecting Garbage -----
found soft-deleted nodes? 1
immediate collect again? 0

----- Hard Display after garb col -----
room
 table
  south west leg
  south east leg
  north west leg
  north east leg
 Lily the canine
  torso
   wagging tail
   left rear paw
   right rear paw
   left front paw
   right front paw
Semi-deleted tree empty? 0

Completely-deleted tree empty? 1


----- Cloned Physical Display Not Affected? -----
room
 table
  south west leg
  south east leg
  north west leg
  north east leg
 Miguel the human (D)
  torso
   right arm
    right hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
   left arm
    left hand
     pinky
     ring finger
     middle finger
     index finger
     thumb
 Lily the canine
  torso
   wagging tail
   spare mutant paw (D)
   left rear paw
   right rear paw
   left front paw
   right front paw
Program ended with exit code: 0

 --------------------------------------------------------------- */
```