```
// Ezekiel Quintanilla
// Assignment 6 Pt 1
//
// CS3358 - Koh

#include <iostream>
#include <cstdlib>
#include "llcpInt.h"
using namespace std;

void SortedMergeRecur (Node*& xNode, Node*& yNode, Node*& zNode)
{
   // If X and Y lists are empty, do nothing
   if (!xNode && !yNode) return;

   if (xNode && !yNode)
   {
      // If Y list is empty but X is not
      // Move X nodes into Z list.
      zNode = xNode;
      xNode = xNode->link;
   }
   else if (!xNode && yNode)
   {
      // If X list is empty but Y is not
      // Move Y nodes into Z list.
      zNode = yNode;
      yNode = yNode->link;
      zNode->link = 0;
   }
   else if ((xNode && yNode) && (xNode->data <= yNode->data))
   {
      // If X and Y have nodes compare their values
      // Add X's node to Z if its smaller
      zNode = xNode;
      xNode = xNode->link;
   }
   else
   {
      // Add Y's node to Z if its smaller
      zNode = yNode;
      yNode = yNode->link;
      zNode->link = 0;
   }

   // Call SortedMergeRecur to iterate through the listsx
   SortedMergeRecur (xNode, yNode, zNode->link);
}

int FindListLength(Node* headPtr)
{
   int length = 0;
```

```
   while (headPtr != 0)
   {
      ++length;
      headPtr = headPtr->link;
   }

   return length;
}

bool IsSortedUp(Node* headPtr)
{
   if (headPtr == 0 || headPtr->link == 0) // empty or 1-node
      return true;
   while (headPtr->link != 0) // not at last node
   {
      if (headPtr->link->data < headPtr->data)
         return false;
      headPtr = headPtr->link;
   }
   return true;
}

void InsertAsHead(Node*& headPtr, int value)
{
   Node *newNodePtr = new Node;
   newNodePtr->data = value;
   newNodePtr->link = headPtr;
   headPtr = newNodePtr;
}

void InsertAsTail(Node*& headPtr, int value)
{
   Node *newNodePtr = new Node;
   newNodePtr->data = value;
   newNodePtr->link = 0;
   if (headPtr == 0)
      headPtr = newNodePtr;
   else
   {
      Node *cursor = headPtr;

      while (cursor->link != 0) // not at last node
         cursor = cursor->link;
      cursor->link = newNodePtr;
   }
}

void InsertSortedUp(Node*& headPtr, int value)
{
   Node *precursor = 0,
        *cursor = headPtr;
```

```
while (cursor != 0 && cursor->data < value)
{
   precursor = cursor;
   cursor = cursor->link;
}

Node *newNodePtr = new Node;
newNodePtr->data = value;
newNodePtr->link = cursor;
if (cursor == headPtr)
   headPtr = newNodePtr;
else
   precursor->link = newNodePtr;

//////////////////////////////////////////////////////
/* using-only-cursor (no precursor) version
Node *newNodePtr = new Node;
newNodePtr->data = value;
//newNodePtr->link = 0;
//if (headPtr == 0)
//   headPtr = newNodePtr;
//else if (headPtr->data >= value)
//{
//   newNodePtr->link = headPtr;
//   headPtr = newNodePtr;
//}
if (headPtr == 0 || headPtr->data >= value)
{
   newNodePtr->link = headPtr;
   headPtr = newNodePtr;
}
//else if (headPtr->link == 0)
//   head->link = newNodePtr;
else
{
   Node *cursor = headPtr;
   while (cursor->link != 0 && cursor->link->data < value)
      cursor = cursor->link;
   //if (cursor->link != 0)
   //   newNodePtr->link = cursor->link;
   newNodePtr->link = cursor->link;
   cursor->link = newNodePtr;
}

////////////////// commented lines removed /////////////////

Node *newNodePtr = new Node;
newNodePtr->data = value;
if (headPtr == 0 || headPtr->data >= value)
{
   newNodePtr->link = headPtr;
   headPtr = newNodePtr;
}
```

```
   else
   {
      Node *cursor = headPtr;
      while (cursor->link != 0 && cursor->link->data < value)
         cursor = cursor->link;
      newNodePtr->link = cursor->link;
      cursor->link = newNodePtr;
   }
   */
   ////////////////////////////////////////////////////////
}

bool DelFirstTargetNode(Node*& headPtr, int target)
{
   Node *precursor = 0,
        *cursor = headPtr;

   while (cursor != 0 && cursor->data != target)
   {
      precursor = cursor;
      cursor = cursor->link;
   }
   if (cursor == 0)
   {
      cout << target << " not found." << endl;
      return false;
   }
   if (cursor == headPtr) //OR precursor == 0
      headPtr = headPtr->link;
   else
      precursor->link = cursor->link;
   delete cursor;
   return true;
}

bool DelNodeBefore1stMatch(Node*& headPtr, int target)
{
   if (headPtr == 0 || headPtr->link == 0 || headPtr->data == target) return false;
   Node *cur = headPtr->link, *pre = headPtr, *prepre = 0;
   while (cur != 0 && cur->data != target)
   {
      prepre = pre;
      pre = cur;
      cur = cur->link;
   }
   if (cur == 0) return false;
   if (cur == headPtr->link)
   {
      headPtr = cur;
      delete pre;
   }
   else
   {
```

```
      prepre->link = cur;
      delete pre;
   }
   return true;
}

void ShowAll(ostream& outs, Node* headPtr)
{
   while (headPtr != 0)
   {
      outs << headPtr->data << "  ";
      headPtr = headPtr->link;
   }
   outs << endl;
}

void FindMinMax(Node* headPtr, int& minValue, int& maxValue)
{
   if (headPtr == 0)
   {
      cerr << "FindMinMax() attempted on empty list" << endl;
      cerr << "Minimum and maximum values not set" << endl;
   }
   else
   {
      minValue = maxValue = headPtr->data;
      while (headPtr->link != 0)
      {
         headPtr = headPtr->link;
         if (headPtr->data < minValue)
            minValue = headPtr->data;
         else if (headPtr->data > maxValue)
            maxValue = headPtr->data;
      }
   }
}

double FindAverage(Node* headPtr)
{
   if (headPtr == 0)
   {
      cerr << "FindAverage() attempted on empty list" << endl;
      cerr << "An arbitrary zero value is returned" << endl;
      return 0.0;
   }
   else
   {
      int sum = 0,
          count = 0;

      while (headPtr != 0)
      {
         ++count;
```

```
         sum += headPtr->data;
         headPtr = headPtr->link;
      }

      return double(sum) / count;
   }
}

void ListClear(Node*& headPtr, int noMsg)
{
   int count = 0;

   Node *cursor = headPtr;
   while (headPtr != 0)
   {
      headPtr = headPtr->link;
      delete cursor;
      cursor = headPtr;
      ++count;
   }
   if (noMsg) return;
   clog << "Dynamic memory for " << count << " nodes freed"
        << endl;
}
```