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


void SortedMergeRecur(Node*& xhead, Node*& yhead, Node*& zhead)
{
    //empty list
    if (xhead == 0 && yhead == 0) return;

    //if the Y-list is empty reassign the X into the new pointer
    else if (xhead != 0 && yhead == 0){
        zhead = xhead;
        zhead->link = xhead->link;
        xhead = xhead->link;
        return SortedMergeRecur(xhead, yhead, zhead->link);
    }

    //if the X-list is empty reassign the Y into the new pointer
    else if (xhead == 0 && yhead != 0){
        zhead = yhead;
        zhead->link = yhead->link;
        yhead = yhead->link;
        return SortedMergeRecur(xhead, yhead, zhead->link);
    }

    //if neither list is empty we compare which is greater and reassign accordingly
    else {
        if (xhead->data <= yhead->data){
            zhead = xhead;
            zhead->link = xhead->link;
            xhead = xhead->link;
            return SortedMergeRecur(xhead, yhead, zhead->link);
        }
        else {
            zhead = yhead;
            zhead->link = yhead->link;
            yhead = yhead->link;
            return SortedMergeRecur(xhead, yhead, zhead->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)
```

```
67            return false;
68         headPtr = headPtr->link;
69      }
70      return true;
71   }
72
73   void InsertAsHead(Node*& headPtr, int value)
74   {
75      Node *newNodePtr = new Node;
76      newNodePtr->data = value;
77      newNodePtr->link = headPtr;
78      headPtr = newNodePtr;
79   }
80
81   void InsertAsTail(Node*& headPtr, int value)
82   {
83      Node *newNodePtr = new Node;
84      newNodePtr->data = value;
85      newNodePtr->link = 0;
86      if (headPtr == 0)
87         headPtr = newNodePtr;
88      else
89      {
90         Node *cursor = headPtr;
91
92         while (cursor->link != 0) // not at last node
93            cursor = cursor->link;
94         cursor->link = newNodePtr;
95      }
96   }
97
98   void InsertSortedUp(Node*& headPtr, int value)
99   {
100     Node *precursor = 0,
101          *cursor = headPtr;
102
103     while (cursor != 0 && cursor->data < value)
104     {
105        precursor = cursor;
106        cursor = cursor->link;
107     }
108
109     Node *newNodePtr = new Node;
110     newNodePtr->data = value;
111     newNodePtr->link = cursor;
112     if (cursor == headPtr)
113        headPtr = newNodePtr;
114     else
115        precursor->link = newNodePtr;
116
117     ////////////////////////////////////////////////////////
118     /* using-only-cursor (no precursor) version
119     Node *newNodePtr = new Node;
120     newNodePtr->data = value;
121     //newNodePtr->link = 0;
122     //if (headPtr == 0)
123     //   headPtr = newNodePtr;
124     //else if (headPtr->data >= value)
125     //{
126     //   newNodePtr->link = headPtr;
127     //   headPtr = newNodePtr;
128     //}
129     if (headPtr == 0 || headPtr->data >= value)
130     {
131        newNodePtr->link = headPtr;
132        headPtr = newNodePtr;
```

```
133     }
134     //else if (headPtr->link == 0)
135     //   head->link = newNodePtr;
136     else
137     {
138        Node *cursor = headPtr;
139        while (cursor->link != 0 && cursor->link->data < value)
140           cursor = cursor->link;
141        //if (cursor->link != 0)
142        //   newNodePtr->link = cursor->link;
143        newNodePtr->link = cursor->link;
144        cursor->link = newNodePtr;
145     }
146
147     ////////////////// commented lines removed /////////////////
148
149     Node *newNodePtr = new Node;
150     newNodePtr->data = value;
151     if (headPtr == 0 || headPtr->data >= value)
152     {
153        newNodePtr->link = headPtr;
154        headPtr = newNodePtr;
155     }
156     else
157     {
158        Node *cursor = headPtr;
159        while (cursor->link != 0 && cursor->link->data < value)
160           cursor = cursor->link;
161        newNodePtr->link = cursor->link;
162        cursor->link = newNodePtr;
163     }
164     */
165     ////////////////////////////////////////////////////////
166  }
167
168  bool DelFirstTargetNode(Node*& headPtr, int target)
169  {
170     Node *precursor = 0,
171          *cursor = headPtr;
172
173     while (cursor != 0 && cursor->data != target)
174     {
175        precursor = cursor;
176        cursor = cursor->link;
177     }
178     if (cursor == 0)
179     {
180        cout << target << " not found." << endl;
181        return false;
182     }
183     if (cursor == headPtr) //OR precursor == 0
184        headPtr = headPtr->link;
185     else
186        precursor->link = cursor->link;
187     delete cursor;
188     return true;
189  }
190
191  bool DelNodeBefore1stMatch(Node*& headPtr, int target)
192  {
193     if (headPtr == 0 || headPtr->link == 0 || headPtr->data == target) return false;
194     Node *cur = headPtr->link, *pre = headPtr, *prepre = 0;
195     while (cur != 0 && cur->data != target)
196     {
197        prepre = pre;
198        pre = cur;
```

```
199         cur = cur->link;
200      }
201      if (cur == 0) return false;
202      if (cur == headPtr->link)
203      {
204         headPtr = cur;
205         delete pre;
206      }
207      else
208      {
209         prepre->link = cur;
210         delete pre;
211      }
212      return true;
213   }
214
215   void ShowAll(ostream& outs, Node* headPtr)
216   {
217      while (headPtr != 0)
218      {
219         outs << headPtr->data << "  ";
220         headPtr = headPtr->link;
221      }
222      outs << endl;
223   }
224
225   void FindMinMax(Node* headPtr, int& minValue, int& maxValue)
226   {
227      if (headPtr == 0)
228      {
229         cerr << "FindMinMax() attempted on empty list" << endl;
230         cerr << "Minimum and maximum values not set" << endl;
231      }
232      else
233      {
234         minValue = maxValue = headPtr->data;
235         while (headPtr->link != 0)
236         {
237            headPtr = headPtr->link;
238            if (headPtr->data < minValue)
239               minValue = headPtr->data;
240            else if (headPtr->data > maxValue)
241               maxValue = headPtr->data;
242         }
243      }
244   }
245
246   double FindAverage(Node* headPtr)
247   {
248      if (headPtr == 0)
249      {
250         cerr << "FindAverage() attempted on empty list" << endl;
251         cerr << "An arbitrary zero value is returned" << endl;
252         return 0.0;
253      }
254      else
255      {
256         int sum = 0,
257             count = 0;
258
259         while (headPtr != 0)
260         {
261            ++count;
262            sum += headPtr->data;
263            headPtr = headPtr->link;
264         }
```

```
265
266         return double(sum) / count;
267      }
268   }
269
270   void ListClear(Node*& headPtr, int noMsg)
271   {
272      int count = 0;
273
274      Node *cursor = headPtr;
275      while (headPtr != 0)
276      {
277         headPtr = headPtr->link;
278         delete cursor;
279         cursor = headPtr;
280         ++count;
281      }
282      if (noMsg) return;
283      clog << "Dynamic memory for " << count << " nodes freed"
284           << endl;
285   }
```