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

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;
```

```
70      newNodePtr->link = cursor;
71      if (cursor == headPtr)
72         headPtr = newNodePtr;
73      else
74         precursor->link = newNodePtr;
75
76      ////////////////////////////////////////////////////////
77      /* using-only-cursor (no precursor) version
78      Node *newNodePtr = new Node;
79      newNodePtr->data = value;
80      //newNodePtr->link = 0;
81      //if (headPtr == 0)
82      //   headPtr = newNodePtr;
83      //else if (headPtr->data >= value)
84      //{
85      //   newNodePtr->link = headPtr;
86      //   headPtr = newNodePtr;
87      //}
88      if (headPtr == 0 || headPtr->data >= value)
89      {
90         newNodePtr->link = headPtr;
91         headPtr = newNodePtr;
92      }
93      //else if (headPtr->link == 0)
94      //   head->link = newNodePtr;
95      else
96      {
97         Node *cursor = headPtr;
98         while (cursor->link != 0 && cursor->link->data < value)
99            cursor = cursor->link;
100        //if (cursor->link != 0)
101        //   newNodePtr->link = cursor->link;
102        newNodePtr->link = cursor->link;
103        cursor->link = newNodePtr;
104     }
105
106     ////////////////// commented lines removed //////////////////
107
108     Node *newNodePtr = new Node;
109     newNodePtr->data = value;
110     if (headPtr == 0 || headPtr->data >= value)
111     {
112        newNodePtr->link = headPtr;
113        headPtr = newNodePtr;
114     }
115     else
116     {
117        Node *cursor = headPtr;
118        while (cursor->link != 0 && cursor->link->data < value)
119           cursor = cursor->link;
120        newNodePtr->link = cursor->link;
121        cursor->link = newNodePtr;
122     }
123     */
124     ////////////////////////////////////////////////////////
125  }
126
127  bool DelFirstTargetNode(Node*& headPtr, int target)
128  {
129     Node *precursor = 0,
130          *cursor = headPtr;
131
132     while (cursor != 0 && cursor->data != target)
133     {
134        precursor = cursor;
135        cursor = cursor->link;
136     }
137     if (cursor == 0)
138     {
```

```
139         cout << target << " not found." << endl;
140         return false;
141      }
142      if (cursor == headPtr) //OR precursor == 0
143         headPtr = headPtr->link;
144      else
145         precursor->link = cursor->link;
146      delete cursor;
147      return true;
148   }
149
150   bool DelNodeBefore1stMatch(Node*& headPtr, int target)
151   {
152      if (headPtr == 0 || headPtr->link == 0 || headPtr->data == target) return false;
153      Node *cur = headPtr->link, *pre = headPtr, *prepre = 0;
154      while (cur != 0 && cur->data != target)
155      {
156         prepre = pre;
157         pre = cur;
158         cur = cur->link;
159      }
160      if (cur == 0) return false;
161      if (cur == headPtr->link)
162      {
163         headPtr = cur;
164         delete pre;
165      }
166      else
167      {
168         prepre->link = cur;
169         delete pre;
170      }
171      return true;
172   }
173
174   void ShowAll(ostream& outs, Node* headPtr)
175   {
176      while (headPtr != 0)
177      {
178         outs << headPtr->data << "  ";
179         headPtr = headPtr->link;
180      }
181      outs << endl;
182   }
183
184   void FindMinMax(Node* headPtr, int& minValue, int& maxValue)
185   {
186      if (headPtr == 0)
187      {
188         cerr << "FindMinMax() attempted on empty list" << endl;
189         cerr << "Minimum and maximum values not set" << endl;
190      }
191      else
192      {
193         minValue = maxValue = headPtr->data;
194         while (headPtr->link != 0)
195         {
196            headPtr = headPtr->link;
197            if (headPtr->data < minValue)
198               minValue = headPtr->data;
199            else if (headPtr->data > maxValue)
200               maxValue = headPtr->data;
201         }
202      }
203   }
204
205   double FindAverage(Node* headPtr)
206   {
207      if (headPtr == 0)
```

```
208      {
209         cerr << "FindAverage() attempted on empty list" << endl;
210         cerr << "An arbitrary zero value is returned" << endl;
211         return 0.0;
212      }
213      else
214      {
215         int sum = 0,
216             count = 0;
217
218         while (headPtr != 0)
219         {
220            ++count;
221            sum += headPtr->data;
222            headPtr = headPtr->link;
223         }
224
225         return double(sum) / count;
226      }
227   }
228
229   void ListClear(Node*& headPtr, int noMsg)
230   {
231      int count = 0;
232
233      Node *cursor = headPtr;
234      while (headPtr != 0)
235      {
236         headPtr = headPtr->link;
237         delete cursor;
238         cursor = headPtr;
239         ++count;
240      }
241      if (noMsg) return;
242      clog << "Dynamic memory for " << count << " nodes freed"
243           << endl;
244   }
245
246   // definition of SortedMergeRecur
247   void SortedMergeRecur(Node*& headX, Node*& headY, Node*& headZ)
248   {
249      if (headX == 0 && headY == 0) //If both X-list and Y-list are empty
250      {
251         return;
252      }
253
254      else if(headX != 0 && headY == 0) //X-list NOT empty, Y-list empty
255      {
256         headZ = headX;
257         headX = 0;
258      }
259
260      else if(headX == 0 && headY != 0) //Y-list NOT empty, X-list empty
261      {
262         headZ = headY;
263         headY = 0;
264      }
265
266      else if(headX != 0 && headY != 0) //X-list NOT empty, Y-list NOT empty
267      {
268         if(headX -> data < headY -> data) //X-list element less than Y-list element
269         {
270              headZ = headX;
271              headX = headX -> link;
272              headZ -> link = 0;
273         }
274
275         else  //Y-list element less than X-list element
276         {
```

```
277             headZ = headY;
278             headY = headY -> link;
279             headZ -> link = 0;
280         }
281 
282         SortedMergeRecur(headX, headY, headZ -> link);
283     }
284 
285 }
286 
```