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

// definition of Merge2AscListsRecur
// (put here to facilitate grading)
void Merge2AscListsRecur(Node* headX, Node* headY, Node* headZ)
{

    if (headX == 0 && headY == 0)
    {
        return;
    }

    if (headX == 0)
    {
        Node*& temp = headY;
        headZ = temp;
        headY = headY->link;

        Merge2AscListsRecur(headX, headY, headZ->link);

        return;
    }

    if (headY == 0)
    {
        Node*& temp = headX;
        headZ = headX;
        headX = headX->link;

        Merge2AscListsRecur(headX, headY, headZ->link);

        return;
    }

     // Find smallest value node
    if (headX->data < headY->data || headX->data == headY->data)
    {
        Node*& temp = headX;
        headZ = temp;
        headX = headX->link;

        Merge2AscListsRecur(headX, headY, headZ->link);
    }
    else
    {
        Node*& temp = headY;
```

```
50          headZ = temp;
51          headY = headY->link;
52
53          Merge2AscListsRecur(headX, headY, headZ->link);
54      }
55
56  }
57
58
59  int FindListLength(Node* headPtr)
60  {
61     int length = 0;
62
63     while (headPtr != 0)
64     {
65        ++length;
66        headPtr = headPtr->link;
67     }
68
69     return length;
70  }
71
72  bool IsSortedUp(Node* headPtr)
73  {
74     if (headPtr == 0 || headPtr->link == 0) // empty or 1-node
75        return true;
76     while (headPtr->link != 0) // not at last node
77     {
78        if (headPtr->link->data < headPtr->data)
79           return false;
80        headPtr = headPtr->link;
81     }
82     return true;
83  }
84
85  void InsertAsHead(Node*& headPtr, int value)
86  {
87     Node *newNodePtr = new Node;
88     newNodePtr->data = value;
89     newNodePtr->link = headPtr;
90     headPtr = newNodePtr;
91  }
92
93  void InsertAsTail(Node*& headPtr, int value)
94  {
95     Node *newNodePtr = new Node;
96     newNodePtr->data = value;
97     newNodePtr->link = 0;
98     if (headPtr == 0)
```

```
 99        headPtr = newNodePtr;
100     else
101     {
102        Node *cursor = headPtr;
103
104        while (cursor->link != 0) // not at last node
105           cursor = cursor->link;
106        cursor->link = newNodePtr;
107     }
108  }
109
110  void InsertSortedUp(Node*& headPtr, int value)
111  {
112     Node *precursor = 0,
113          *cursor = headPtr;
114
115     while (cursor != 0 && cursor->data < value)
116     {
117        precursor = cursor;
118        cursor = cursor->link;
119     }
120
121     Node *newNodePtr = new Node;
122     newNodePtr->data = value;
123     newNodePtr->link = cursor;
124     if (cursor == headPtr)
125        headPtr = newNodePtr;
126     else
127        precursor->link = newNodePtr;
128
129     ////////////////////////////////////////////////////////
130     /* using-only-cursor (no precursor) version
131     Node *newNodePtr = new Node;
132     newNodePtr->data = value;
133     //newNodePtr->link = 0;
134     //if (headPtr == 0)
135     //   headPtr = newNodePtr;
136     //else if (headPtr->data >= value)
137     //{
138     //   newNodePtr->link = headPtr;
139     //   headPtr = newNodePtr;
140     //}
141     if (headPtr == 0 || headPtr->data >= value)
142     {
143        newNodePtr->link = headPtr;
144        headPtr = newNodePtr;
145     }
146     //else if (headPtr->link == 0)
147     //   head->link = newNodePtr;
```

```
148     else
149     {
150        Node *cursor = headPtr;
151        while (cursor->link != 0 && cursor->link->data < value)
152           cursor = cursor->link;
153        //if (cursor->link != 0)
154        //   newNodePtr->link = cursor->link;
155        newNodePtr->link = cursor->link;
156        cursor->link = newNodePtr;
157     }
158
159     ////////////////// commented lines removed /////////////////
160
161     Node *newNodePtr = new Node;
162     newNodePtr->data = value;
163     if (headPtr == 0 || headPtr->data >= value)
164     {
165        newNodePtr->link = headPtr;
166        headPtr = newNodePtr;
167     }
168     else
169     {
170        Node *cursor = headPtr;
171        while (cursor->link != 0 && cursor->link->data < value)
172           cursor = cursor->link;
173        newNodePtr->link = cursor->link;
174        cursor->link = newNodePtr;
175     }
176     */
177     ////////////////////////////////////////////////////////
178  }
179
180  bool DelFirstTargetNode(Node*& headPtr, int target)
181  {
182     Node *precursor = 0,
183          *cursor = headPtr;
184
185     while (cursor != 0 && cursor->data != target)
186     {
187        precursor = cursor;
188        cursor = cursor->link;
189     }
190     if (cursor == 0)
191     {
192        cout << target << " not found." << endl;
193        return false;
194     }
195     if (cursor == headPtr) //OR precursor == 0
196        headPtr = headPtr->link;
```

```
197     else
198        precursor->link = cursor->link;
199     delete cursor;
200     return true;
201  }
202
203  bool DelNodeBefore1stMatch(Node*& headPtr, int target)
204  {
205     if (headPtr == 0 || headPtr->link == 0 || headPtr->data == target)
          return false;
206     Node *cur = headPtr->link, *pre = headPtr, *prepre = 0;
207     while (cur != 0 && cur->data != target)
208     {
209        prepre = pre;
210        pre = cur;
211        cur = cur->link;
212     }
213     if (cur == 0) return false;
214     if (cur == headPtr->link)
215     {
216        headPtr = cur;
217        delete pre;
218     }
219     else
220     {
221        prepre->link = cur;
222        delete pre;
223     }
224     return true;
225  }
226
227  void ShowAll(ostream& outs, Node* headPtr)
228  {
229     while (headPtr != 0)
230     {
231        outs << headPtr->data << "  ";
232        headPtr = headPtr->link;
233     }
234     outs << endl;
235  }
236
237  void FindMinMax(Node* headPtr, int& minValue, int& maxValue)
238  {
239     if (headPtr == 0)
240     {
241        cerr << "FindMinMax() attempted on empty list" << endl;
242        cerr << "Minimum and maximum values not set" << endl;
243     }
244     else
```

```
245     {
246        minValue = maxValue = headPtr->data;
247        while (headPtr->link != 0)
248        {
249           headPtr = headPtr->link;
250           if (headPtr->data < minValue)
251              minValue = headPtr->data;
252           else if (headPtr->data > maxValue)
253              maxValue = headPtr->data;
254        }
255     }
256  }
257
258  double FindAverage(Node* headPtr)
259  {
260     if (headPtr == 0)
261     {
262        cerr << "FindAverage() attempted on empty list" << endl;
263        cerr << "An arbitrary zero value is returned" << endl;
264        return 0.0;
265     }
266     else
267     {
268        int sum = 0,
269            count = 0;
270
271        while (headPtr != 0)
272        {
273           ++count;
274           sum += headPtr->data;
275           headPtr = headPtr->link;
276        }
277
278        return double(sum) / count;
279     }
280  }
281
282  void ListClear(Node*& headPtr, int noMsg)
283  {
284     int count = 0;
285
286     Node *cursor = headPtr;
287     while (headPtr != 0)
288     {
289        headPtr = headPtr->link;
290        delete cursor;
291        cursor = headPtr;
292        ++count;
293     }
```

```
294     if (noMsg) return;
295     clog << "Dynamic memory for " << count << " nodes freed"
296          << endl;
297 }
298
```